João Miguel Lobo Fernandes

MIDAS: Metodologia Orientada ao Objecto para Desenvolvimento de Sistemas Embebidos

Tese submetida à Escola de Engenharia da Universidade do Minho
para a obtenção do grau de Doutor em Informática
(Área de Conhecimento em Engenharia de Computadores)

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Braga — Fevereiro 2000

João Miguel Lobo Fernandes

MIDAS: Metodologia Orientada ao Objecto para Desenvolvimento de Sistemas Embebidos

Tese submetida à Escola de Engenharia da Universidade do Minho
para a obtenção do grau de Doutor em Informática,
Área de Conhecimento em Engenharia de Computadores

Dissertação realizada sob a co-orientação do
Prof. Doutor Alberto José Gonçalves de Carvalho Proença,
Professor Associado com Agregação do Departamento de Informática da
Escola de Engenharia da Universidade do Minho
e do
Prof. Doutor Henrique Manuel Dinis dos Santos,
Professor Auxiliar do Departamento de Sistemas de Informação da
Escola de Engenharia da Universidade do Minho

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Braga — Fevereiro 2000

Título: MIDAS — Metodologia Orientada ao Objecto para Desenvolvimento de Sistemas Embebidos

Autor: João Miguel Lobo Fernandes

Tese de Doutoramento em Informática, Especialidade em Engenharia de Computadores, Departamento de Informática, Escola de Engenharia, Universidade do Minho

Candidatura a Doutoramento aceite pelo Conselho Científico da Escola de Engenharia da Universidade do Minho em 08/Fev/1995

Orientadores: Alberto José Gonçalves de Carvalho Proença e Henrique Manuel Dinis dos Santos

Conclusão: Fevereiro de 2000

À memória de meu pai.

"*Quando no decurso de uma noite nos luz assim de súbito uma ideia, em busca da qual andávamos havia muito, quando nos ocorre a solução de um problema em que meditávamos, impacienta-nos o imperturbável silêncio e quitação que nos rodeia, formando tão completo contraste com o tumulto que nos vai no pensamento. Ansiamos pelo dia para ter a quem comunicar a descoberta, e para examinar à luz bem clara, e desenganamo-nos de que não fomos vítimas de uma ilusão nocturna.*"

Júlio Dinis in "Os Fidalgos da Casa Mourisca".

Prefácio

Reflexões Pessoais

Uma tese de doutoramento não é vista, por mim, como uma espécie de título nobiliárquico que o respectivo autor pode invocar para legitimar todas as suas atitudes de carácter científico. É óbvio que constitui uma promoção (carreira, social, monetária) mas representa, sobretudo, um reconhecimento das nossas capacidades científicas e intelectuais e do nosso poder de análise e síntese; contudo não significa que tudo sabemos. A minha postura é mais humilde, pois julgo que o gosto por aprender e o saber ouvir (e ler) os outros devem sempre acompanhar-nos, se queremos continuar actualizados.

Comparando a investigação científica com a condução de automóveis, a obtenção de um doutoramento corresponde à obtenção da carta de condução: sabemos conduzir, meter as mudanças e estacionar o carro, mas fazemo-lo com muitos cuidados: ainda não nos devemos atrever a andar a grandes velocidades ou a sacar uns piões. Só com a experiência de condução (após muitos quilómetros) é que nos devemos lançar noutro tipo de manobras. Assim, este trabalho de doutoramento representa, para mim, apenas mais um passo na minha trajectória académica e profissional: não é "o" passo e, muito menos, a meta, mas antes e só "um" passo importante e decisivo num percurso, que espero, longo e frutífero.

Durante o período de realização deste trabalho, ocorreram algumas transformações, revoluções e "guerras" dentro do DI/UM, o que nem sempre me permitiu concentrar as minhas energias na investigação. A ida, mais ou menos forçada, dum dos meus co-orientadores para Guimarães provocou um corte muito grande nas actividades que desenvolvíamos conjuntamente e, sobretudo, reduziu o contacto quase diário que tínhamos anteriormente a alguns contactos esporádicos. Foi, sem dúvidas, um factor que muito me penalizou.

Por outro lado, os reduzidos interesse e apoio que senti em relação ao meu trabalho, por parte dos órgãos que gerem as minhas actividades pedagógicas e científicas, foram um factor extremamente desmotivador. O estilo de investigação centrado em "capelinhas" (pequenos grupos fechados) não permite aproveitar sinergias que existem potencialmente no DI/UM e no Algoritmi. O facto de, por vezes, não conhecermos o que os colegas de outros grupos fazem e, consequentemente, não podermos discutir, num plano mais alargado que o grupo, as nossas ideias obriga, na prática, a procurar lá fora aquilo que temos, seguramente, cá em casa.

Agradecimentos

Um trabalho desta natureza, não podia ser completado sem a participação, voluntária ou não, de outras pessoas, além de mim. Pretendo aqui deixar o meu agradecimento a todos aque-

les que mais directamente estiveram envolvidos, confessando que será, com certeza, impossível enumerá-los de modo exaustivo. Aos esquecidos, apresento o meu antecipado pedido de desculpas.

Em primeiro lugar, tenho que reconhecer que este trabalho beneficiou do contributo científico dado pelo Professor Doutor Alberto José Proença, na qualidade de orientador cientifico. Todos os seus sábios conselhos e as suas oportunas sugestões foram, por mim, sempre entendidos como contribuições que me permitiram validar as diversas questões que esta tese pretende abordar.

Agradeço ao Professor Doutor Henrique Dinis Santos, com quem afortunadamente trabalho desde 1991, o apoio que deu à realização desta tese. A incondicional disponibilidade que sempre demonstrou e, ainda, todos os ensinamentos e estímulos que me deu, os quais valorizaram enormemente este projecto, contribuíram decididamente para o trajecto que foi seguido neste trabalho.

Ao meu colega Eng. Ricardo Jorge Machado, tenho de expressar a minha profunda gratidão pela forma entusiasta e profissional com que se envolveu no projecto de investigação que serviu de base aos nossos doutoramentos. Não tenho quaisquer dúvidas em afirmar que, sem a sua participação e a sua constante necessidade em aplicar a nossa metodologia em ambientes industriais, este trabalho teria tomado um rumo menos interessante.

Ao IDITE-Minho, nas pessoas do Eng. Eduardo Pinto e do Eng. Adelino Silva, quero manifestar a minha gratidão pela forma interessada com que me apoiaram no desenvolvimento da aplicação e na possibilidade de introduzir num ambiente real a metodologia Midas. Desejo igualmente formular o meu agradecimento ao Eng. Jorge Cruz Tavares, ao Eng. Francisco Duarte, ao Eng. Fernando Barbosa e ao Eng. José Cunha da Blaupunkt Auto-Rádios de Portugal pela forma dedicada e profissional como trabalharam no projecto de diagnóstico e optimização das linhas de produção. Fico ainda grato aos meus alunos (Celeste Pinto, Paula Monteiro, Filipe Pereira e Hugo Paredes), cujo empenho nos projectos em que conjuntamente nos envolvemos permitiu adiantar algumas das actividades previstas nesta tese.

Aos Docentes, Investigadores, Técnicos e Funcionários do DI/UM, com quem tive a oportunidade de interagir, o meu muito obrigado. Igualmente agradeço ao Professor Doutor Guilherme Pereira e ao Eng. Luís Silva Dias (DPS/UM), a colaboração no âmbito da simulação estatística de processos e na orientação técnica às animações de regras.

Não posso esquecer a minha família mais próxima, especialmente, a minha mãe Elvira e a minha irmã Rita, pela forma carinhosa como sempre me apoiaram, durante o período que durou o projecto. Em particular, queria dedicar este trabalho à memória de meu pai Manuel, cuja postura de vida e gosto pelo conhecimento foram um estímulo que procurei sempre seguir, ao longo da minha vida, e que me permitiu que tivesse um percurso académico e profissional, de que muito me orgulho.

À Raquel, minha mulher, amiga e companheira de vida, deixo uma palavra de eterna gratidão por todo o apoio, amor e carinho incondicionais, sem os quais este projecto não teria sido concluído.

Finalmente, ao meu filho Gonçalo que, apesar de ainda não ter nascido, foi também uma fonte de estímulo para que o trabalho fosse realizado com mais empenho e ânimo.

Braga, 16 de Fevereiro de 2000.

Resumo

O desenvolvimento metódico e sistemático de sistemas embebidos apresenta-se, nos dias de hoje, como uma disciplina muito importante para a economia dum país civilizado, devido ao forte impacto que aquela classe de sistemas induz em diversas áreas industriais, nomeadamente no controlo, monitorização e supervisão de processos fabris.

Nesta tese, apresenta-se a metodologia MIDAS que pretende auxiliar o projectista no desenvolvimento de sistemas embebidos, partindo do pressuposto que estes serão implementados em plataformas mistas (com hardware e software). O trabalho foca essencialmente as questões associadas à análise e à modelação de sistemas embebidos complexos, nomeadamente o levantamento de requisitos, os meta-modelos a usar para especificação e os métodos a seguir, não abordando, duma forma tão profunda, questões associadas às fases de concepção e implementação.

Os pilares da metodologia MIDAS são: (1) a modelação orientada ao objecto; (2) o recurso a especificações unificadas, gráficas e multi-vista; (3) a adopção da abordagem operacional. Neste sentido, mostram-se as vantagens que a adopção da abordagem orientada ao objecto pode trazer à modelação de sistemas embebidos e a importância em recorrer a especificações unificadas (por não estarem comprometidas com a solução final), gráficas (por serem claras e intuitivas, mas, simultaneamente, precisas e rigorosas) e multi-vista (por representarem diferentes vistas do mesmo sistema) para lidar com sistemas complexos. A possibilidade em executar as especificações (abordagem operacional) e a adopção dum processo iterativo e incremental são dois factores cruciais para o desenvolvimento de sistemas de relativa complexidade, motivo pelo qual se incentiva o seu uso em MIDAS.

É apresentado um meta-modelo multi-vista apropriado para a especificação de sistemas embebidos e são seleccionadas as respectivas notações, que se baseiam integralmente na notação UML. Indica-se, também, o modelo de processo e, para cada uma das tarefas que o compõem, é apresentado o correspondente método, composto por algumas recomendações, que podem ser utilizadas para a execução da referida tarefa.

Finalmente, com o objectivo de validar e demonstrar a aplicação da metodologia MIDAS, são descritos, sob o ponto de vista técnico e metodológico, 2 projectos de desenvolvimento de casos reais (um sistema de supervisão de iluminação e o sistema de controlo das linhas de produção duma fábrica).

Palavras-chave: sistemas embebidos, metodologias de desenvolvimento, análise de sistemas, orientação ao objecto, UML.

Abstract

The systematic and methodical development of embedded systems is nowadays a very important discipline for the economy of an industrial country. These systems have a strong impact on several industrial application domains, namely in control, monitoring and supervision of manufacturing processes.

The main result of this thesis is a new methodology to design embedded systems: MIDAS. This methodology presents a unified view to develop mixed platform solutions, containing both hardware and software components. The work focus mainly on the issues related to the analysis and modelling of complex embedded systems, with special emphasis on the requirements capture task, the specification meta-models, and the methods to be followed. The design and implementation phases are less detailed.

The MIDAS methodology is based on the following topics: (1) object-oriented modelling; (2) unified, graphical, and multiple-view specifications; (3) an operational approach. It is shown how the object-oriented paradigm has advantages to model embedded systems, and the relevance of using specifications that are: unified (to not compromise the final solution), graphical (to be intuitive and clear, but simultaneously, precise and rigorous) and multiple-viewed (to represent different views of the same system) to manage the system's complexity. When developing relatively complex systems, some critical issues are pertinent and the MIDAS methodology is strongly recommended; these issues include the direct execution of the specifications (operational approach) and an approach towards an iterative and incremental process.

A multiple-view meta-model is presented to appropriately specify embedded systems and the corresponding notations, which are completely based on the UML notation are indicated. The process model is also presented and for each task that belongs to it, the corresponding method is described. The methods description include guidelines that can be used to accomplish the referred task.

Finally, to validate and show how to apply the MIDAS methodology, 2 case studies are described (a lighting supervision system and the control system of an industrial production line), from the technical and methodological point of view.

Keywords: embedded systems, development methodologies, systems' analysis, object-orientation, UML.

Conteúdo

Lista de Figuras

Lista de Tabelas

Parte I

Conceitos nucleares

Capítulo 1

Introdução

Antes não começar que não acabar.
Ter começado é meio caminho andado.

Sumário

Este capítulo inicia-se com uma caracterização, em linhas gerais, do tipo de sistemas considerados neste trabalho, os sistemas embebidos. De seguida, apresenta-se uma série de definições para termos de uso generalizado, ao longo do texto, e indicam-se os principais motivos que determinam o uso de técnicas orientadas ao objecto para desenvolver sistemas embebidos. Depois, são enumerados os principais objectivos que se pretendem alcançar com esta tese e fazem-se algumas referências ao enquadramento em que este trabalho foi realizado. A finalizar, é descrita a forma como esta tese foi estruturada.

Índice

1.1 Sistemas embebidos

Um *sistema baseado em computadores* (que também pode designar-se *sistema computacional*) é definido como um conjunto de elementos que são organizados para atingir um objectivo pré-definido, através do processamento automático de informação. Para alcançar o objectivo estabelecido, um sistema computacional utiliza os seguintes tipos de elementos [Pressman, 1997, pág. 248]:

- **Software**: Programas e estruturas de dados que realizam a funcionalidade requerida.
- **Hardware**: Dispositivos electrónicos que disponibilizam a capacidade computacional e os dispositivos electro-mecânicos que providenciam o contacto com o exterior.
- **Pessoas**: Utilizadores e operadores que utilizam o hardware e o software.
- **Base de dados**: Repositório de informação que é usado através do software.
- **Documentação**: Manuais, relatórios e outros documentos que descrevem o uso e/ou a operação do sistema.
- **Procedimentos**: Passos que definem o uso específico de cada elemento do sistema ou o contexto no qual este reside.

Neste trabalho, os sistemas computacionais a estudar incluem, sobretudo, a classe dos *sistemas embebidos*, que são sistemas concebidos para uma aplicação específica, contendo componentes de hardware e de software [Kumar et al., 1996b]. Regra geral, um sistema embebido faz parte, como o próprio nome indicia, dum sistema mais complexo e nele se encontra incorporado (embutido, embebido). O termo "embebido", que foi popularizado pelo Departamento de Defesa do governo estadunidense, refere-se ao facto de esses sistemas estarem embebidos (isto é, incluídos) em sistemas maiores, cuja principal função não é a computação [Zave, 1982]. Neste sentido, e de acordo com a definição de sistema dada na pág. 7, um sistema será visto como embebido, quando o seu observador perspectivar a sua utilização no contexto dum sistema ou ambiente mais alargado.

Os sistemas embebidos costumam ser construídos para controlar um dado ambiente físico, interactuando directamente com dispositivos eléctricos e indirectamente com dispositivos mecânicos (fig. 1.1). Para tal, um sistema embebido recorre a um conjunto de sensores e a uma série de actuadores que lhe permite, respectivamente, receber informação relativamente ao ambiente e pôr em prática as respostas elaboradas pelo sistema, para enviar ao ambiente.

A classe dos sistemas embebidos representa uma fatia muito importante dos sistemas com que todos lidamos diariamente, tanto em casa, como principalmente em ambientes industriais. Exemplos típicos de sistemas embebidos são os sistemas de controlo de automóveis, de electrodomésticos, de impressoras ou de máquinas industriais; os sistemas de aquisição de dados para equipamento laboratorial; os sistemas de apoio ao diagnóstico médico; ou os sistemas de controlo e monitorização de processos industriais.

As propriedades mais relevantes que caracterizam os sistemas embebidos são as que a seguir se enumeram [Camposano e Wilberg, 1996]:

- Um sistema embebido é, normalmente, desenvolvido para realizar uma **função específica** para uma dada aplicação. Muitas vezes, será apenas posto a funcionar um único sistema, o que indica que, em princípio, um sistema embebido não será produzido em massa.
- Usualmente é requerido que o sistema tenha um **funcionamento em contínuo**, i.e. que esteja a operar enquanto também estiver a funcionar o sistema mais amplo.
- Os sistemas embebidos devem manter uma **interacção permanente** com o respectivo ambiente, ou seja, devem responder continuamente a diferentes eventos provenientes do

exterior e cuja ordem e tempo de ocorrência não são, regra geral, previsíveis.

- A **correcção** dum sistema embebido é um factor crucial, pois, habitualmente, os sistemas embebidos realizam tarefas críticas em termos de fiabilidade e segurança, pelo que um único erro pode representar danos graves ou mesmo perda de vidas humanas em situações mais extremas.
- Os sistemas embebidos devem, por vezes, obedecer a imposições ou restrições temporais, pelo que questões de **tempo-real** têm que ser equacionadas. O típico sistema embebido é um sistema de tempo-real[1].
- Os sistema electrónicos, em geral, e os sistemas embebidos, em particular, são, na sua generalidade, **digitais**.

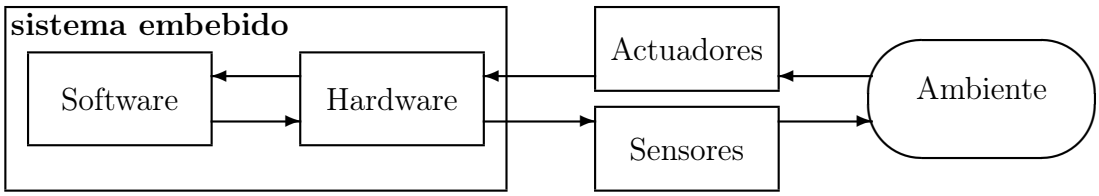


Figura 1.1: A estrutura típica dum sistema embebido.

Dadas estas propriedades, parece ser aceitável que o termo "sistema embebido" possa, no âmbito deste trabalho, incluir, entre outros, os sistemas reactivos, os sistemas de monitorização, os sistemas de controlo (de processos) e os sistemas de tempo-real.

Um *sistema reactivo* é aquele que mantém uma interacção permanente ou frequente com o seu ambiente e que responde continuamente, em função do seu estado interno, aos estímulos externos a que está sujeito. Os sistemas reactivos não podem ser descritos apenas pela especificação dos sinais de saída em função dos sinais de entrada, mas, antes, por especificações que relacionam as entradas e as saídas ao longo do tempo. Tipicamente, as descrições de sistemas reactivos incluem sequências de eventos, acções, condições e fluxos de informação, combinadas, frequentemente, com restrições temporais, para definir o comportamento global do sistema. Em terminologia da área dos sistemas digitais, um sistema reactivo é classificado como sistema sequencial e não como sistema combinatório. Um *sistema combinatório* é formado por funções lógicas elementares, tem um conjunto de entradas e saídas, e os valores destas dependem exclusivamente do valor das entradas e da constituição interna do sistema. Num *sistema sequencial*, os valores das saídas, num dado momento, não dependem exclusivamente dos valores aplicados nas entradas, nesse instante, mas também dos valores que estavam presentes anteriormente. Pode acontecer que, para combinações iguais nas entradas, se obtenham valores distintos nas saídas, em momentos diferentes [Padilla, 1993, pág. 123]. Pode também acrescentar-se que os sistemas sequenciais contêm a noção de memória ou estado, contrariamente ao que sucede com os sistemas combinatórios.

Os *sistemas de monitorização* e os *sistemas de controlo* são responsáveis pela supervisão dum dado ambiente não "inteligente", informando constantemente o seu utilizador (humano ou não) do estado daquele e actuando em situações críticas. Por exemplo, numa instalação industrial, quando um dado gás atinge uma determinada concentração (pré-indicada), o respectivo sistema de controlo deve imediatamente accionar um alarme.

[1]Como referido por Zave: "*Embedded is almost synonymous with real-time.*" [Zave, 1982].

Um *sistema de tempo-real* é um sistema reactivo, cujo comportamento deve respeitar, além da funcionalidade pretendida, um conjunto de restrições temporais externamente definidas [Benveniste e Berry, 1991]. Um aspecto crítico dum sistema de tempo-real, como o próprio nome denuncia, é a forma como as questões temporais são tratadas. No desenvolvimento dum sistema deste tipo, além de ser necessário satisfazer a sua correcção em termos funcionais, devem identificar-se os requisitos temporais e deve assegurar-se que estes são cumpridos durante o desempenho do sistema.

É habitual classificarem-se os sistemas de tempo-real em termos dos tempos de reacção que exibem relativamente às necessidades do meio em que se enquadram. Neste sentido, as restrições temporais a que os sistemas podem estar sujeitos, dividem-se em três categorias principais, correspondendo cada uma delas a um tipo distinto de sistemas[2]:

- **Sistemas de tempo-real forte**[3]: A correcção duma resposta inclui também o instante em que é dada. Uma resposta atrasada é incorrecta e constitui uma falha do sistema. Por exemplo, o tempo limite para a resposta dum alarme a uma situação de perigo deve ser escrupulosamente cumprido, caso contrário pode não ser possível salvar o sistema.
- **Sistemas de tempo-real fraco**[4]: Os requisitos desta categoria são especificados com um valor médio de resposta. Se uma resposta chega atrasada, não ocorre uma falha no sistema, antes apenas uma degradação do desempenho do sistema. Por exemplo, se um ficheiro demorar mais tempo a ser aberto do que o inicialmente previsto, o utilizador pode ficar insatisfeito ou desesperado, conforme a sua expectativa quanto ao nível de desempenho do sistema, mas desse facto não resulta qualquer falha.
- **Sistemas de tempo-real frouxo**[5]: Trata-se duma situação de compromisso entre as duas categorias anteriores, em que é tolerada uma baixa probabilidade no incumprimento dum tempo de resposta que leva a uma falha do sistema.

De acordo com esta categorização, o termo "tempo-real", quando usado isoladamente, refere-se, neste trabalho, à categoria forte.

Para terminar, sempre que se usar, de aqui em diante, o termo embebido, para classificar um sistema, está implicitamente a referir-se a um sistema reactivo que tem de cumprir requisitos temporais (rígidos ou não) e que pode ser usado para controlo ou monitorização dum determinado processo ou sistema.

1.2 Taxinomia e definições

Uma das consequências inevitáveis da rápida evolução da computação e, em particular, da área do desenvolvimento metodológico de sistemas consiste na utilização frequente, por diferentes pessoas, da mesma palavra com conotações diversas. Como forma de evitar esta situação, de seguida, dar-se-ão definições para uma série de termos que terão uso reiterado ao longo deste trabalho. Não se teve a veleidade de definir os termos de raiz: houve o cuidado de comparar alguma terminologia usada e depois fez-se uma selecção daquela que pareceu mais adequada. É muito provável que o leitor não considere o resultado final como o mais apropriado, mas tal facto só virá, na opinião do autor, demonstrar a pertinência deste estudo taxinómico. A ausência

[2]Para classificar os vários tipos de sistema de tempo-real, foram usadas as traduções sugeridas por Gomes para os termos em língua inglesa, como pode ser verificado nas três notas de rodapé seguintes [Gomes, 1997].

[3]Tempo-real forte como tradução de *hard real-time*.

[4]Tempo-real fraco como tradução de *soft real-time*.

[5]Tempo-real frouxo como tradução de *firm real-time*.

desta secção seria, com toda a certeza, um mal bem maior, porque deixaria a interpretação ou o significado dos termos dependentes da experiência e da área de trabalho do leitor, que não são necessariamente idênticas às do autor.

Refira-se que a definição de terminologia foi uma necessidade sentida no seio do grupo de investigação a que o autor pertence, devido às distintas formações dos seus elementos. Durante algumas discussões internas, era evidente que certos termos eram interpretados de forma diferenciada, o que conduzia, muitas vezes, a desviar o centro da discussão para questões de taxinomia, em detrimento dos assuntos que interessava realmente debater.

1.2.1 Hardware e software

Duas das palavras mais usadas, ao longo deste trabalho, são *hardware* e *software*, pelo que são fornecidas as definições sugeridas por Tanenbaum para cada uma delas [Tanenbaum, 1990, pág. 11].

Os circuitos electrónicos dum sistema computacional, bem como a memória e os dispositivos de entrada/saída formam o hardware do sistema. O *hardware* dum sistema é constituído por objectos tangíveis (circuitos integrados, *chips*, placas de circuito impresso, cabos, fontes de alimentação, memórias, impressoras, terminais) e não por ideias abstractas, algoritmos ou instruções.

Por seu lado, o *software* dum sistema consiste em *algoritmos* (instruções descrevendo pormenorizadamente como fazer algo) e respectivas representações, nomeadamente os *programas*. Um determinado programa pode ser representado em diversos suportes (cartão perfurado, fita magnética, disquete, disco compacto, etc.), mas a sua essência reside no conjunto de instruções que o constitui e não no meio físico no qual é armazenado.

Uma forma intermédia entre o hardware e o software é o *firmware*, que consiste em software embebido em dispositivos electrónicos durante a sua manufactura [Tanenbaum, 1990, pág. 11]. O firmware é usado quando se espera, por exemplo, que os programas nunca sejam alterados (ou pelo menos que o sejam raramente) ou quando os programas não se podem perder em caso de falta de fornecimento de energia. Em alguns processadores, o seu funcionamento é controlado por um *microprograma*, que é firmware.

1.2.2 Sistema, modelo e especificação

Um *sistema* pode ser definido como uma colecção de componentes inter-relacionados que actuam como um todo, para atingir um determinado objectivo. Esta definição permite que praticamente tudo o que existe no universo seja entendido como um sistema, o que acaba por ser verdade, pois é difícil imaginar algo que não possa ser, potencialmente, visto como um sistema[6]. Contudo, para o engenheiro, o que interessa não é saber se algo é ou não um sistema, mas antes se esse "algo" é, por ele, visto como um sistema [Thomé, 1993, pág. 5]. Quando tal se verifica, é porque há um interesse da sua parte em estudar as propriedades desse sistema como um todo. É o observador do sistema que define a fronteira deste com o seu ambiente, o que torna a definição de sistema não intrínseca a este, mas dependente do seu observador, ou seja, dos objectivos particulares deste em cada situação. Como consequência, os componentes que,

[6]Neste contexto, um electrão, se considerado como uma partícula elementar (sem componentes), não pode ser visto como um sistema.

num determinado contexto, constituem um dado sistema, podem, noutro, ser apenas um sub-sistema dum sistema mais amplo. Como exemplo dum sistema, que será usado para auxiliar a compreender as definições dadas a seguir, considere-se uma elevador para transporte vertical de pessoas e bens.

Uma *vista do sistema* é uma perspectiva (total ou parcial) do sistema que se pretende representar. Por exemplo, a descrição dum livro pode ser feita segundo duas vistas distintas [Sigfried, 1996, pág. 29]. Uma hipótese é descrever o livro como sendo constituído por várias páginas; cada página é descrita como uma série de linhas, podendo eventualmente incluir figuras; cada linha, por sua vez, é composta por símbolos (letras, dígitos, pontuação). Outra vista, mais lógica, baseia-se na estrutura do conteúdo do livro. Um livro está, segundo essa perspectiva, dividido em capítulos; cada um dos capítulos tem um ou mais sub-capítulos; um sub-capítulo pode conter várias secções; finalmente, chega-se aos parágrafos, em que cada um é composto por diversas frases, que, por sua vez, se compõem de palavras. As palavras, por seu lado, são constituídas por letras.

O sistema controlado (i.e. os meios mecânicos, as portas, os botões, os sensores, etc.) ou a unidade de controlo do elevador são duas vistas distintas do mesmo sistema. Outra vista possível é a estrutural em que se descreve a constituição física do sistema; no caso do elevador, essa vista indica a localização do motor, o número de pisos, as dimensões das portas, etc.

Um *meta-modelo* (modelo dum modelo) é um conjunto de elementos, funcionais ou estruturais, de composição e de regras de composição que permitem construir uma representação conceptual (virtual ou abstracta) para o sistema. Como exemplos, considerem-se os seguintes meta-modelos: redes de Petri, máquinas de estados ou, grafos de fluxo de dados e controlo. O meta-modelo deve ser *formal* (preciso, rigoroso), evitando ambiguidades na interpretação da representação do sistema, e deve também ser *completo*, permitindo a construção duma representação que descreva totalmente o sistema. Estas duas características do meta-modelo não são absolutas, mas, como já foi referido nas definições, dependem do sistema considerado. Quando um meta-modelo é formal, pode designar-se por *formalismo*.

Um *modelo* é uma representação conceptual (virtual ou abstracta) do sistema, à luz dum determinado meta-modelo. Exemplos de modelos são a máquina de estados da unidade de controlo do elevador ou o grafo de fluxo de dados e controlo desse elevador. O modelo manifesta todas as características do respectivo meta-modelo, ou seja, quando o meta-modelo é formal, o modelo também o é, e quando o meta-modelo é completo, o modelo também é completo[7]. Um modelo diz-se *formal* (preciso, rigoroso) quando evita ambiguidades na sua interpretação e *completo* quando representa totalmente o sistema.

Admite-se a hipótese dum projectista obter modelos que não sejam formais e/ou completos por, consciente ou inconscientemente, ter seleccionado um meta-modelo que não possua essas características; a manifestação dessas características no modelo obtido deve-se unicamente ao meta-modelo seleccionado.

O *meta-modelo da linguagem* é o modelo conceptual, seguido na definição da linguagem e implica que qualquer especificação escrita nessa linguagem segue obrigatoriamente o meta-modelo por ela imposto. O meta-modelo da linguagem OCCAM é aquele que é definido pela notação CSP [Hoare, 1978]. A linguagem CONPAR, definida no âmbito do mestrado do autor [Fernandes, 1994], tem por meta-modelo a RdP-SI (acrónimo de Rede de Petri Síncrona e

[7]Parte-se do pressuposto que o projectista explora e usa todas as características do meta-modelo na construção do modelo.

Interpretada). Todas as afirmações anteriormente referidas no âmbito da definição de meta-modelo aplicam-se igualmente ao meta-modelo da linguagem, por se tratar também, com toda a propriedade, dum meta-modelo.

A *gramática duma linguagem* define-se à custa duma sintaxe, dum léxico e duma semântica que permitem concretizar, à luz do meta-modelo da linguagem, o modelo do sistema numa representação gráfica, textual ou outra.

Uma *linguagem* (ou *notação*) é o conjunto de todas as frases válidas que é possível construir utilizando a respectiva gramática. Por exemplo, VHDL e Pascal são duas linguagens. A linguagem manifesta todas as características do seu meta-modelo, ou seja, se o meta-modelo da linguagem é formal então a linguagem é formal e, similarmente, se o meta-modelo da linguagem é completo, então a linguagem também é completa.

Uma linguagem para especificação de sistemas embebidos deve possuir as seguintes características:

- **Formal** (precisa, rigorosa): Evitando ambiguidades na especificação do modelo do sistema.
- **Completa**: Permitindo obter uma especificação que descreve totalmente o modelo pretendido para o sistema.
- **Executável**: Possibilitando a validação do modelo do sistema.

Espera-se ainda que a linguagem promova a facilidade de utilização e modificação da especificação do modelo do sistema.

Uma *especificação* é uma representação concreta (real) do modelo do sistema numa dada linguagem. Como exemplos, têm-se máquinas de estados da unidade de controlo do elevador escritas em VHDL ou em C. As características da especificação dependem simultaneamente das características da linguagem (ou por arrasto do meta-modelo da linguagem) e do modelo do sistema (e inevitavelmente do meta-modelo do sistema). Em termos práticos, não há grandes diferenças, ao nível conceptual, entre uma especificação e um modelo dum dado sistema, pois ambos são representações deste, pelo que, por vezes, os termos especificação e modelo são usados, de forma relaxada, como sinónimos.

A obtenção duma especificação que represente adequadamente o sistema depende das características do meta-modelo utilizado no modelo do sistema, bem como do meta-modelo da linguagem que a suporta. Desta forma, deve seleccionar-se uma linguagem cujo respectivo meta-modelo seja o mais idêntico possível ao meta-modelo do sistema, para evitar "incompatibilidades" na especificação[8]. É incompatível especificar um modelo RdP-SI dum controlador paralelo, em linguagem Fortran, uma vez que esta não inclui quaisquer mecanismos que permitam explicitar a concorrência que os modelos RdP-SI potencialmente podem possuir. Contudo, já é compatível especificar, em linguagem ConPar, um modelo RdP-SI dum controlador, pois esta linguagem foi especificamente concebida para suportar todos os elementos e regras de composição do meta-modelo RdP-SI.

Em consonância com o que foi anteriormente escrito, *modelar* é o acto que permite a obtenção de modelos dum sistema, enquanto que *especificar* corresponde à criação de especificações para

[8]Considera-se que há incompatibilidades no processo de especificação, quando a linguagem não possui os construtores ou os mecanismos indispensáveis para modelar, fácil e directamente, o sistema em questão. Incompatibilidade pode, numas situações, querer significar dificuldade ou inadequabilidade e noutras pode mesmo expressar impossibilidade. Uma analogia que pode ser indicada é a utilização de ferramentas (meta-modelo) para apertar (especificar) um parafuso. Com um martelo é praticamente impossível; com um canivete é difícil; com um desandador, se correctamente utilizado, é simples.

os modelos dum sistema (utilizando linguagens).

A fig. 1.2 pretende relacionar e estratificar alguns dos termos anteriormente definidos, onde se pode constatar que os meta-modelos podem ser utilizados indiscriminadamente em relação a linguagens e a modelos.

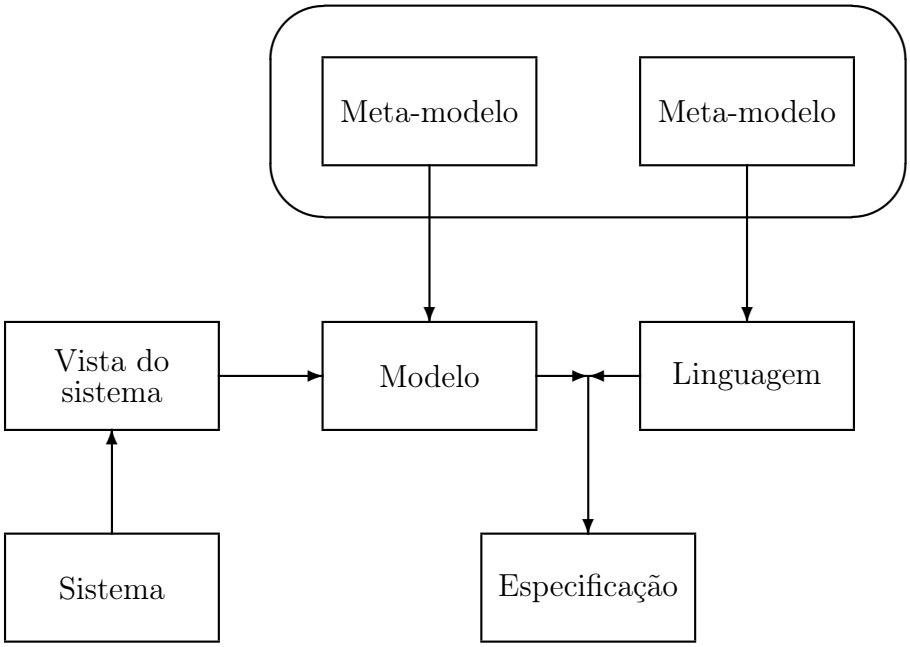


Figura 1.2: Os meta-modelos e a sua relação com as linguagens e os modelos, usados na especificação dum sistema.

1.2.3 Ciclo de vida, desenvolvimento e metodologia

Em sistemas de software, usa-se o termo *ciclo de vida* (*life cycle*) para referir o conjunto de actividades que se inicia no momento em que o sistema é mentalmente conceptualizado (assim que surge a ideia, a necessidade ou a vontade de o construir) até ao instante em que ele é retirado definitivamente de uso (quando é colocado na sucata). Por outras palavras, o ciclo de vida representa todo o conjunto de actos válidos, realizados, durante a vida útil do sistema, com o objectivo de o idealizar, desenvolver e usar.

O termo *desenvolvimento* refere-se às fases do ciclo de vida responsáveis pela construção do sistema, incluindo a análise, a concepção e a implementação. Excluem-se, por exemplo, os estudos de viabilidade económica, as tarefas de manutenção e a utilização efectiva do sistema. O ciclo de vida do sistema inclui algumas fases pré-desenvolvimento (estudos económicos de viabilidade) e outras pós-desenvolvimento (manutenção ou reengenharia).

O termo *projecto* é usado, neste trabalho, com o significado de representar as actividades anteriores à utilização prática do respectivo sistema. *Projectista* ou, em sentido mais abrangente, *equipa de projecto* são os termos usados para indicar os indivíduos responsáveis por qualquer das fases do projecto dum dado sistema (ou por uma parte deste). Por vezes, podem usar-se termos como *analista* ou *programador* para referir um projectista especialista numa dada fase do projecto. Do mesmo modo, os termos *utilizador* e *cliente* são usados para indicar, respectivamente, o indivíduo que interage directamente com o sistema final e a pessoa que encomenda

o desenvolvimento do sistema à equipa de projecto. Apesar desta distinção entre utilizador e cliente, neste trabalho, esses termos são usados indistintamente, uma vez que, muitas vezes, esses dois papéis são desempenhados pela mesma pessoa.

A definição supra indicada para utilizador é bastante limitadora, já que é possível distinguir vários tipos de utilizadores. Para um avião, por exemplo, os diferentes tipos de utilizadores[9] podem ser: os passageiros, os pilotos, os hospedeiros, o pessoal da manutenção, os técnicos de controlo de voo, os empregados de limpeza, os formadores de pilotagem, etc.

Em sentido geral, chama-se *processo* à sequência de factos que conduzem a certo resultado (exemplos: processo de fabrico, processo químico, processo judicial), mais particularmente, à sequência de actos ordenados para a consecução de certa finalidade. Por exemplo, o processo industrial, numa dada fábrica, representa o conjunto de actos executados para produzir um determinado bem de consumo, desde a chegada das matérias primas até à embalagem para carregamento. No âmbito da construção metódica de sistemas, processo significa o conjunto de tarefas executadas ao longo do ciclo de vida do sistema. O termo *processo de desenvolvimento* tem um âmbito menos abrangente e restringe-se às tarefas realizadas durante o desenvolvimento do sistema.

Um *modelo de processo* é um esquema que organiza, ordena e relaciona a forma como as várias fases e tarefas devem ser prosseguidas ao longo do ciclo de vida do sistema. A função principal dum modelo de processo é determinar a ordem das fases envolvidas no desenvolvimento de sistemas e estabelecer os critérios de transição para progredir entre fases [Boehm, 1988]. Similarmente, pode considerar-se a existência dum *modelo de processo de desenvolvimento*, com iguais restrições às indicadas no parágrafo anterior.

Pode então colocar-se a seguinte questão: "quais as diferenças entre um processo e o respectivo modelo?". Um modelo pode, por exemplo, considerar a existência de duas tarefas que, potencialmente, poderão ser executadas em paralelo. No entanto, durante a real execução dum processo, que segue esse modelo, essas referidas tarefas são executadas em sequência (qualquer ordem), enquanto que noutro processo são, realmente, executadas em paralelo.

Uma *metodologia* incentiva uma dada abordagem para desenvolver um sistema, através da selecção dum conjunto de métodos (ou técnicas) a serem aplicados. O propósito duma metodologia é, pois, promover uma determinada abordagem, para resolver um dado problema, mediante a prévia escolha dos métodos e técnicas a serem usados [Ghezzi et al., 1991, pág. 43]. Uma metodologia pode ser apresentada como uma série de passos, aos quais estão associadas técnicas e notações [Rumbaugh et al., 1991, pág. 144]. Apesar de alguns autores usarem o termo paradigma como sinónimo de metodologia [Morris et al., 1996, pág. 22], neste trabalho, optou-se por diferenciar os dois termos. Assim, um *paradigma* representa um conjunto de teorias, técnicas e métodos que, conjuntamente, representam uma forma de organizar a informação, ou seja, uma paradigma é uma forma de representar a realidade.

Uma *fase* é uma abstracção, ao longo do tempo, dum conjunto de actividades, ou seja, é um conceito útil para agregar actividades e relacioná-las temporalmente [Whytock, 1993]. O ciclo de vida dum sistema complexo é normalmente dividido nas seguintes fases genéricas[10]:

- **Estudo de viabilidade**: O propósito desta fase consiste na avaliação dos custos e bene-

[9] O termo utilizador é, segundo esta perspectiva mais alargada, usado para representar as pessoas que influenciam o sistema, como sinónimo da palavra inglesa *stakeholder*.

[10] Embora não se pressuponha nenhum modelo de processo em especial, pode, por simplicidade conceptual, assumir-se um processo segundo, por exemplo, o modelo em cascata (ver subsecção 2.2.1) em que há uma divisão mais explícita entre as fases do que a que se verifica noutros modelos.

fícios do sistema proposto.

- **Análise**: Nesta fase, são levantados os requisitos do sistema e produz-se um modelo abstracto que descreve os aspectos fundamentais do domínio de aplicação e que permite captar a essência do sistema em causa.
- **Concepção**: Com base no modelo obtido na fase de análise, é criado um modelo que especifica os componentes que realizam uma determinada solução para o sistema.
- **Implementação**: Realiza-se uma solução, descrita numa dada linguagem de programação, transformando-se o modelo de concepção.
- **Teste** (fase guarda-chuva): Durante o desenvolvimento do sistema, esta fase decorre em paralelo, a fim de se tentar encontrar todas as falhas que o sistema possa conter.
- **Utilização**: Operação do sistema em ambiente real.
- **Manutenção** (fase guarda-chuva): Durante a utilização do sistema, esta fase procura corrigir todas as anomalias que não foi possível detectar durante o desenvolvimento e pretende também fazer evoluir o sistema de modo a que continue a ser útil aos seus utilizadores.

A fig. 1.3 pretende relacionar e mostrar o âmbito de alguns dos termos definidos nesta subsecção.

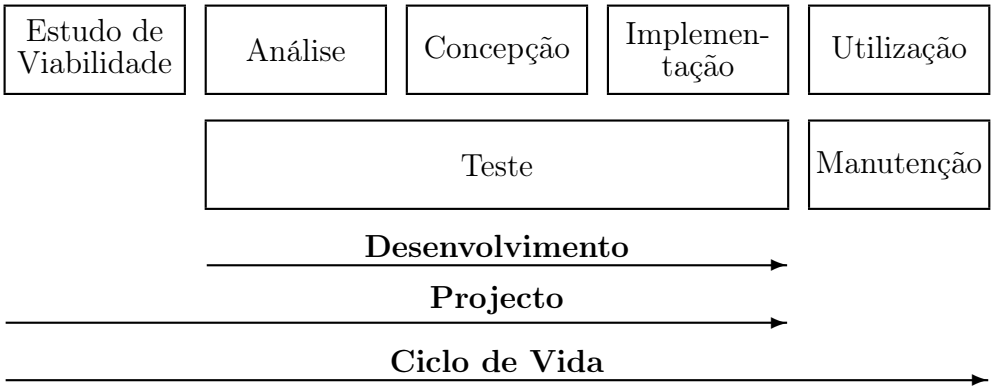


Figura 1.3: Relação dos termos ciclo de vida, projecto e desenvolvimento com as várias fases que um sistema atravessa.

Um *método* é um conjunto de actividades (recomendações genéricas) que organiza a execução duma dada fase do ciclo de vida. Neste trabalho, considera-se que um método é um modo de prosseguir com uma dada fase de desenvolvimento. Existem métodos que se debruçam exclusivamente sobre a fase de análise [Coad e Yourdon, 1991], métodos de concepção [Budgen, 1994] e mesmo métodos de teste [Kit, 1995]. A seguinte definição, sugerida por Morris *et al.*, para método pode também considerar-se como adequada para este trabalho:

> "*A systematic way of proceeding with a well defined phase of development of a computer system product. A method is composed of a series of steps.*" [Morris et al., 1996, pág. 22].

Um nota de particular atenção é, aqui, necessária, pois o termo "método" é, por vezes, usado, na engenharia de software, como sinónimo de função, rotina, procedimento ou serviço. Na linguagem SMALLTALK, um método é uma função que um dado objecto (ou classe de objectos) está apto a realizar. Porém, a distinção de qual o significado preciso é facilmente obtida mediante o contexto em que o termo é usado.

Por vezes, há grandes dúvidas relativamente às diferenças substantivas entre metodologia e método. Assim, não será de estranhar que aquilo, que alguns autores chamam método, possa aqui ser referido como metodologia e *vice-versa*. Duma forma simplista, pode afirmar-se que um método é uma maneira de proceder, duma dada forma ordenada, enquanto que uma metodologia pode definir-se como um conjunto de métodos. Booch distingue claramente método e metodologia, com as seguintes definições dos termos:

> "*A method is a disciplined process for generating a set of models that describe various aspects of a software system under development, using some well-defined notation. A methodology is a collection of methods applied across the software development life cycle and unified by some general, philosophical approach.*" [Booch, 1991, pág. 18].

A perspectiva de Ghezzi *et al.* é muita idêntica à de Booch, apesar de introduzirem o conceito de técnica para relacionarem o método e a metodologia.

> "*Methods are general guidelines that govern the execution of some activity: they are rigorous, systematic and disciplined approaches. Techniques are more technical and mechanical than methods; (...) methods and techniques are packaged together to form a methodology.*" [Ghezzi et al., 1991, pág. 43].

1.3 Motivação

Na construção dum sistema embebido, é prática comum o desenvolvimento independente das suas componentes de hardware e de software, seguido da interligação, que se supõe ser simples e fácil, dessas duas componentes. Devido à maleabilidade do software[11], é crença generalizada que as deficiências encontradas no hardware podem ser sempre rectificadas pelo software. Os processos tradicionais de desenvolvimento seguem um fluxo idêntico ao apresentado na fig. 1.4, em que o desenvolvimento do hardware e o desenvolvimento do software são separados, numa fase inicial do projecto. As duas componentes são desenvolvidas independentemente uma da outra, com pouca interacção, até se chegar à fase de integração.

Embora, potencialmente, o modelo de processo permita o desenvolvimento em simultâneo do hardware e do software, tal prática não é usual. É mais frequente definir, em primeiro lugar, completamente uma das componentes, e só depois a outra. Neste sentido, um método comummente seguido no desenvolvimento dum sistema embebido consiste em basear a definição da sua funcionalidade na componente software, sendo o hardware visto unicamente como a plataforma de suporte à execução do software. Esta abordagem, centrada no software, representa a perspectiva típica dum engenheiro de software. Alternativamente, pode seguir-se uma abordagem centrada no hardware, em que se considera a definição deste como primordial, ao qual, depois, há que adicionar o software para completar a funcionalidade do sistema. Esta é a abordagem habitual do engenheiro de hardware e, neste caso, o hardware é especificado sem apreciar devidamente os requisitos computacionais do software em termos, por exemplo, da velocidade de execução ou da capacidade de memória. O resultado é que o desenvolvimento do software, em

[11] O software é considerado mais maleável (i.e. mais adaptável, mais facilmente alterável) do que o hardware por ter um nível de programabilidade muito elevado. Todavia, a vulgarização da lógica programável e os crescentes avanços da computação reconfigurável fazem prever que, num futuro não muito distante, o hardware atinja um grau de maleabilidade idêntico ao observável, hoje em dia, no software.

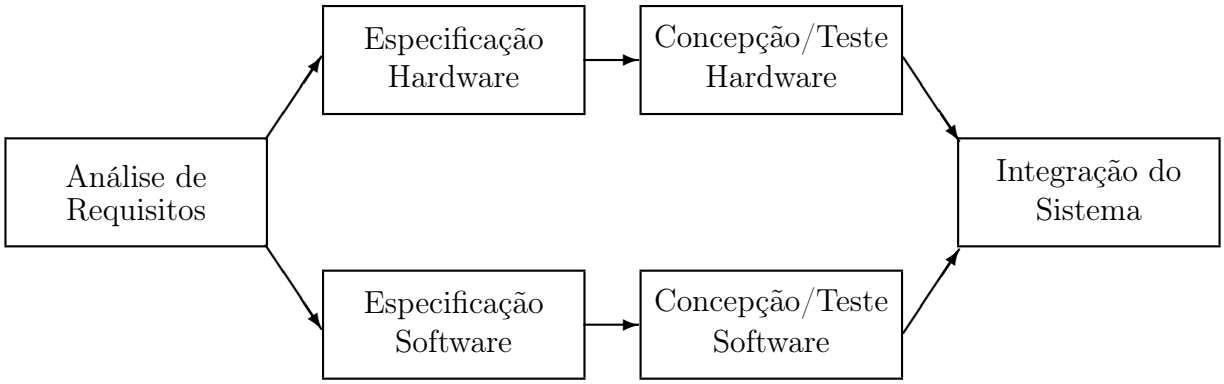


Figura 1.4: O modelo de processo tradicional de desenvolvimento de sistemas embebidos.

princípio, não vai permitir alterar o hardware, pois este já está definido. Em ambos os casos, é somente na fase de interligação que as duas componentes são integradas e tratadas como um todo.

O uso de processos que seguem um modelo semelhante ao esquematizado na fig. 1.4 apresenta diversas consequências negativas. Por um lado, devido à interligação tardia das duas componentes, problemas encontrados nesta fase exigem modificações tanto no hardware como no software, as quais provocam um aumento significativo nos custos e um maior tempo de desenvolvimento. É sabido que problemas detectados durante a fase de integração têm um "custo" de resolução muito superior, se comparados com aquele que resultaria se os erros fossem detectados durante fases mais iniciais do projecto. Por outro lado, a transferência de funcionalidades entre os domínios do hardware e do software e a modificação da interface não são facilitadas. Esta importante restrição invalida que se possam ir corrigindo as deficiências encontradas ao longo do projecto do sistema.

O desenvolvimento de sistemas embebidos complexos, recorrendo a soluções mistas de hardware e software, conheceu enormes avanços nos últimos anos, mas ainda não existe uma metodologia sólida e universalmente aceite que permita o projecto dum sistema, entendido como um todo uno e não como um somatório de partes, as quais terão de ser interligadas de alguma maneira. Pode, portanto, concluir-se que é necessário definir novas metodologias que permitam desenvolver, dum modo mais simples, sistemático, metódico e sustentado, os sistemas embebidos a implementar em plataformas contendo hardware e software. Estas novas metodologias enquadram-se na área do co-projecto hardware/software, que pode ser definido como uma abordagem unificada para o desenvolvimento de sistemas digitais a implementar em plataformas mistas.

1.4 Objectivos do trabalho

O objectivo principal deste trabalho consiste na verificação da validade em se utilizar a abordagem orientada ao objecto para modelar sistemas embebidos (ver secção 1.1), a fim de serem implementados em plataformas mistas (com hardware e software). Dado que a abordagem orientada ao objecto já é usada há alguns anos, com largo sucesso, no desenvolvimento de sistemas software, foram as metodologias da engenharia de software que serviram como grande fonte de inspiração ao autor deste trabalho, para desenvolver sistemas embebidos, contendo

hardware e software. Assim, o co-projecto de hardware/software é visto como uma extensão ou derivação da engenharia de software, mas tendo o cuidado de enquadrar a sua utilização no desenvolvimento integrado de sistemas contendo, simultaneamente, software e hardware.

A utilização da metodologia proposta para desenvolvimento de exemplos reais (um sistema de supervisão de iluminação e o sistema de controlo das linhas de produção duma fábrica) é incluída, nesta tese, com o intuito de demonstrar a aplicabilidade da primeira (ver capítulo 7).

O principal foco desta tese relaciona-se com a problemática da modelação de sistemas embebidos de elevada complexidade, especialmente as questões relacionadas com os meta-modelos a adoptar e os métodos a seguir. Neste sentido, as questões ligadas às fases de concepção e implementação foram tratadas a um nível de profundidade muito menor. É ainda de referir que, neste trabalho, vários assuntos ligados ao desenvolvimento de sistemas (estudo de viabilidade, gestão de projecto, controlo da qualidade, partição, teste, entre outros) não foram abordados duma forma sistematizada, pois, apesar da sua extrema pertinência, não se enquadravam nos objectivos que foram inicialmente definidos.

Propositada e realisticamente foi desenvolvido um conjunto mínimo de utilitários de suporte à metodologia adoptada, não porque se considere pouco importante a existência dum pacote o mais completo possível e respectiva utilidade, mas por a ênfase desta tese se centrar principalmente nas questões teóricas e metodológicas associadas ao desenvolvimento de sistemas embebidos e na aplicação real da metodologia a alguns casos práticos.

Resumidamente, podem indicar-se os seguintes objectivos a cumprir neste trabalho:

- Utilizar a abordagem orientada ao objecto na modelação dos sistemas.
- Seleccionar o meta-modelo mais adequado para especificação de sistemas embebidos e escolher as respectivas notações.
- Definir uma metodologia de desenvolvimento para o projecto de sistemas embebidos.
- Dar um contributo mais profundo na definição da fase de análise da metodologia, sem contudo esquecer as fases de concepção, implementação e teste.
- Aplicar a metodologia a casos práticos.
- Usar o utilitário Oblog para especificar os sistemas e gerar as implementações (em C/C++ e VHDL).

1.4.1 A postura na investigação

A fig. 1.5 sintetiza a postura relativamente à forma como a investigação foi tomada neste trabalho para definição duma metodologia de desenvolvimento de sistemas embebidos. A figura salienta a estreita cumplicidade entre teoria, utilitários e aplicações.

Esta visão em relação à investigação é nitidamente uma visão de engenharia; foram as necessidades sentidas, ao longo do percurso profissional do autor, simultaneamente nos papéis de investigador e de projectista, que conduziram à procura de novas soluções para os problemas que foram surgindo. Neste contexto, pode afirmar-se que as aplicações funcionaram como mola impulsionadora de todo o processo. Houve contudo o cuidado de não desequilibrar a balança, pelo que foi dada igual importância às outras duas componentes. Por um lado, o estudo de quais os meta-modelos e os métodos mais adequados e, por outro, a construção/utilização de utilitários foram também considerados neste trabalho.

Pode concluir-se que a visão tri-partida, identificada na fig. 1.5, se revelou positiva, uma vez que os avanços numa das áreas condicionou, no bom sentido do termo, as outras. Os progressos

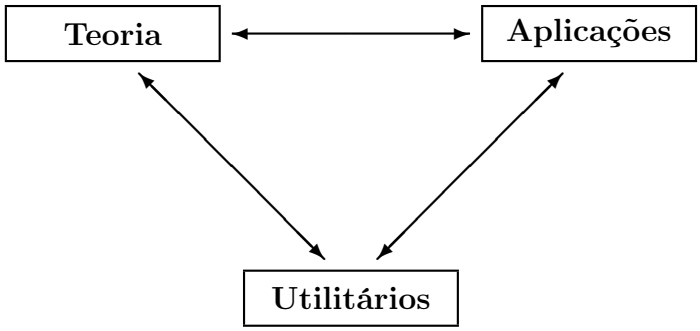


Figura 1.5: A importância da teoria, das aplicações e dos utilitários para o desenrolar da investigação.

em cada uma das três áreas beneficiaram dos conhecimentos que se foram adquirindo nas outras duas, o que potenciou o aparecimento de sinergias que de outro modo seriam muito difíceis de alcançar.

1.5 Enquadramento do trabalho

Este trabalho insere-se nas actividades de investigação levadas a cabo pelo grupo de Engenharia de Computadores, do Departamento de Informática da Escola de Engenharia da Universidade do Minho (DI/UM), sendo enquadrado cientificamente na linha de Informática do centro ALGORITMI. Este grupo, de que é responsável o Prof. Alberto José Proença, tem como principais áreas de interesse, no âmbito da Investigação e Desenvolvimento, as arquitecturas paralelas e avançadas, a visão por computador e os sistemas digitais.

A dissertação de mestrado do autor [Fernandes, 1994] inseriu-se dentro da linha de trabalho do grupo, tendo sido desenvolvidos um meta-modelo e uma linguagem, que se designam, respectivamente, por RdP-SI (Redes de Petri Síncronas e Interpretadas) e CONPAR (Controladores Paralelos), para especificação e validação de sistemas de controlo. Este trabalho teve, depois, continuidade numa outra dissertação de mestrado [Machado, 1996], onde foram acrescentados mecanismos orientados ao objecto para modelação de sistemas digitais, de que resultou o meta-modelo RdP-shobi (Redes de Petri Síncronas, Hierárquicas, Orientadas ao objecto e Interpretadas), permitindo modelar não só a unidade de controlo, como anteriormente já acontecia, mas também o sistema controlado (*data path*).

Uma vez que a utilização, em alguns casos práticos, dos meta-modelos supra indicados mostrou claramente algumas limitações, que urgia colmatar, esta tese de doutoramento surge assim como uma evolução natural relativamente ao trajecto anteriormente percorrido, na medida em que propõe uma metodologia de âmbito mais amplo, que utiliza meta-modelos mais poderosos que os inicialmente adoptados.

O trabalho de investigação foi realizado no âmbito do projecto PRAXIS-XXI "Sistemas Embebidos Reconfiguráveis", cujo objectivo consistia na definição duma metodologia de desenvolvimento (análise, concepção e implementação) para sistemas embebidos com requisitos de funcionamento em tempo-real, recorrendo a lógica reconfigurável para satisfazer esses requisitos, bem como diminuir o respectivo tempo de projecto.

Nome do projecto	Sistemas Embebidos Reconfiguráveis: Metodologias de Desenvolvimento para Aplicações Tempo-Real
Investigador responsável	Prof. Henrique Dinis Santos
Duração	1999-2000 (2 anos)
Instituições envolvidas	Centro Algoritmi (Univ. Minho)
	INESC-Lisboa
	Blaupunkt Auto-Rádio Portugal, Lda.
	IDITE-Minho
	Oblog Software S.A.
Financiamento	PRAXIS-XXI

1.6 Considerações linguísticas

Num trabalho com uma forte componente técnica como é este, é mais ou menos inevitável o recurso a termos em línguas estrangeiras e, em especial, em inglês. Tentou usar-se, sempre que possível, terminologia em português, mas a inexistência dum organismo governamental que estabeleça quais os termos "oficiais" a empregar, obrigou, muitas vezes, por uma questão de clareza, a optar ou pelo termo em língua estrangeira ou então a indicar, simultaneamente, uma tradução em português e o respectivo termo original.

O uso do género masculino não deve ser entendido como uma escolha machista ou sexista, mas deve-se tão somente a razões históricas da língua portuguesa. Quando é usado, por exemplo, o termo "engenheiro" está a referir-se indistintamente a um homem ou a uma mulher que possua as habilitações e as acreditações para exercer engenharia. Seria muito penalizante, em termos da simplicidade pretendida para o texto, ter que escrever "o(a) engenheiro(a)".

Houve o cuidado extremo de eliminar qualquer erro gramatical ou tipográfico do texto final, mas, infelizmente, é muito provável que tal objectivo não tenha sido totalmente conseguido.

1.7 Conteúdo e organização da dissertação

Esta tese foi estruturalmente dividida em 8 capítulos.

O presente capítulo faz uma breve introdução às temáticas tratadas nesta tese e apresenta os objectivos e enquadramento do trabalho realizado.

Os dois capítulos seguintes tentam fazer um retrato do panorama actual do desenvolvimento integrado de sistemas (hardware/software), segundo a abordagem dos objectos. O cap. 2 ("desenvolvimento de sistemas") trata os diversos modelos de processo que podem ser seguidos num projecto dum sistema e mostra que o hardware e o software têm muitas características comuns, o que permite que abordagens unificadas e inicialmente descomprometidas da tecnologia de implementação possam ser postas em prática. No cap. 3 ("modelação de sistemas com objectos") define-se o termo objecto que surge com o significado de componente, seja ele hardware ou software, dum sistema e apresenta-se uma série de conceitos e características associadas aos objectos. Faz-se ainda uma apresentação dos diversos mecanismos de modelação de UML, que é uma linguagem de modelação concebida originalmente para especificar, visualizar, construir

e documentar sistemas software, mas que pode ser igualmente aplicada na modelação de outros tipos de sistemas, nomeadamente, sistemas embebidos a implementar em plataformas hardware e software.

Os 4 capítulos seguintes descrevem o contributo intelectual que consubstancia esta tese. No cap. 4 ("a metodologia MIDAS") mostra-se a necessidade em seguir uma dada metodologia para desenvolver, com menores riscos de falha, sistemas complexos. Nesse sentido, propõe-se a metodologia MIDAS, cujos elementos basilares são: abordagem operacional; especificação unificada, gráfica e multi-vista; e modelação orientada ao objecto. No cap. 5 ("a análise na metodologia MIDAS") é tratado, com grande profundidade, o método a seguir na fase de análise e indicam-se os vários documentos (alguns escritos segundo a notação UML) a criar para cumprir as tarefas que compõem esta fase. O cap. 6 ("a representação unificada OBLOG") trata a construção duma especificação unificada para sistemas embebidos. Foi seleccionada a linguagem OBLOG, devido aos mecanismos de modelação que apresenta e pela facilidade em adaptá-la a vários ambientes de implementação. No cap. 7 ("validação da metodologia proposta") apresentam-se dois exemplos em que se aplicou a metodologia MIDAS, para mostrar, com casos reais, as potencialidades, mas simultaneamente, as limitações da metodologia.

A tese termina com o cap. 8, em que se tecem algumas conclusões sobre o trabalho realizado e se apontam alguns rumos possíveis para o prosseguir.

A escrita desta tese prolongou-se por 4 anos, pelo que algumas das secções que a compõem podem não incluir, por motivos óbvios, os mais recentes progressos das respectivas áreas de conhecimento. As referências, nomeadamente os anos de publicação, podem servir de guia para determinar quando foram maioritariamente escritas essas secções.

Capítulo 2

Desenvolvimento de Sistemas

Quem ler leia para saber; quem souber saiba para obrar.
Quem mais sabe mais aprende.

Sumário

Este capítulo inicia-se com uma descrição da abordagem que tradicionalmente é seguida no desenvolvimento de sistemas embebidos, a que se segue um levantamento relativamente às abordagens e metodologias seguidas no desenvolvimento de software e de hardware. São também apresentadas as características principais das metodologias estruturadas e orientadas ao objecto usadas no desenvolvimento de software e são referidos os objectivos, as actividades e as técnicas associados à fase de análise. Para concluir, é feita uma analogia entre o desenvolvimento nos dois domínios (software e hardware), indicando-se as semelhanças e diferenças entre ambos, e, como consequência, é apresentado o co-projecto de hardware/software.

Índice

2.1 Abordagem tradicional

Há poucos anos, o desenvolvimento dum sistema embebido consistia essencialmente na utilização dum microprocessador de 8 bits (por exemplo, Zilog Z80 ou Intel 8051), que deveria ser cuidadosamente programado por engenheiros com reconhecidas capacidades de programação no *assembly* desse processador [Wilson, 1987]. Actualmente, esta realidade tem vindo a alterar-se, devido à crescente complexidade que os sistemas embebidos têm vindo a adquirir.

O desenvolvimento de sistemas embebidos é considerado mais problemático, quando comparado com o desenvolvimento de sistemas a implementar só em software ou só em hardware, devido à natureza heterogénea dos primeiros, o que requer a concepção e a implementação do hardware e do software, de forma integrada. A maior parte dos sistemas reactivos são altamente concorrentes, o que implica, em algumas situações não consideradas, comportamentos imprevisíveis ou até catastróficos [Harel et al., 1990].

Os sistemas embebidos exigem no seu desenvolvimento, para além do seu correcto funcionamento, que um conjunto de métricas seja cumprido. Ao desenvolvimento de sistemas embebidos está associado um conjunto de requisitos não funcionais que incluem, entre outros, fiabilidade, desempenho e aspectos físicos (tamanho, peso ou consumo).

Os sistemas embebidos podem ser divididos em duas grandes classes, conforme o tipo de desempenho exigido [Cook, 1991]. Sistemas cujo desempenho tem de ser baixo ou médio têm habitualmente requisitos apertados em termos de custos e aspectos físicos. Exemplos deste tipo de sistemas são os controladores para aplicações domésticas: máquinas de lavar, fornos, etc. Para este género de aplicação, o uso de microcontroladores revelava-se, ainda há pouco tempo, mais do que suficiente. Porém, tal realidade tem vindo a alterar-se e já vai sendo, por vezes, difícil "colocar" todo o sistema embebido no microcontrolador escolhido. Um *microcontrolador* é um microprocessador com circuitos auxiliares, tais como conversores A/D, temporizadores, RAM e EPROMs, já incorporados no próprio *chip*. Os fabricantes de microcontroladores tentam, na medida do possível, que estes disponibilizem a máxima funcionalidade, evitando assim que seja necessário acrescentar mais circuitos, uma vez que, se tal se verificar, o custo do sistema será superior.

Sistemas, cujo desempenho tem de apresentar níveis elevados, requerem, regra geral, microprocessadores mais poderosos e, em algumas situações, é mesmo necessário o recurso a soluções envolvendo vários processadores [Wilson, 1989]. Exemplos deste tipo de sistemas são os que se podem encontrar em impressoras, que devem transformar descrições, por vezes complexas, de páginas em mapas de bits, ou em radares, que incluem processamento "inteligente" para detecção de objectos.

Uma abordagem normalmente seguida, por um engenheiro de desenvolvimento de hardware, na produção dum sistema embebido é a seguinte (cf. fig. 1.4):

- Escolha dum microprocessador (ou microcontrolador) comercial. Na esmagadora maioria dos casos, esta opção é ditada principalmente pelo conhecimento específico da equipa de projecto no referido processador (decorrente de outros projectos anteriores), e não, como se impunha, pela adequabilidade do processador para os requisitos do sistema em causa.
- Desenvolvimento duma versão *bread boarded* do hardware a projectar, sob o qual o software irá ser, posteriormente, desenvolvido.
- Em seguida, o projecto do hardware e a produção de software decorrem paralela e independentemente. Como resultado, surge uma primeira versão do hardware final.
- Depois, a equipa de projecto integra o software e o hardware desenvolvidos. Esta fase

produz invariavelmente muitos problemas e nela são detectadas inúmeras falhas, no que resultam alterações, por vezes profundas, tanto na componente hardware, como na componente software.
- Finalmente, após várias iterações e reformulações, é possível construir um protótipo do sistema em desenvolvimento.

A abordagem acima descrita não produz, muito provavelmente, um sistema optimizado (quaisquer que sejam as métricas usadas), o tempo de desenvolvimento é muito prolongado, a manutenção do sistema é difícil, e, não raras vezes, o produto final não responde, infelizmente, aos requisitos do utilizador.

Esta abordagem foi propositadamente seguida pelo autor, conjuntamente com outros elementos do DI/UM, no projecto dum controlador de rega para fins agrícolas, a fim de verificar os problemas que lhe estão associados. Foi usada uma placa de circuito impresso, contendo um Transputer como microprocessador programável, um *watchdog*, um visor, vários relés e memória externa. A componente hardware foi definida em primeiro lugar, tendo depois a restante funcionalidade sido obtida, seguindo uma decomposição funcional, através da escrita de código em linguagem Occam para executar no Transputer.

Entre outros, confirmaram-se alguns dos problemas anteriormente mencionados:
- Análise do problema muito superficial (pouco cuidada).
- Falta de documentação dos diversos elementos do projecto.
- Codificação pouco metódica.
- Processo de integração do software no hardware muito "doloroso".
- Detecção de erros não facilitada.
- Impossibilidade de alterar o hardware (i.e. passar funcionalidades do software para o hardware, depois da placa estar feita).
- Manutenção muito difícil.

Esta experiência foi extremamente enriquecedora, no sentido de que dotou o autor da sensibilidade necessária em relação às inúmeras questões, directa ou indirectamente, ligadas ao projecto de sistemas embebidos. Concomitantemente, mostrou a indiscutível necessidade de definir uma metodologia, cuja ênfase seja posta nas fases de análise e de concepção do sistema, antes de definir o tipo de tecnologia a usar para implementar, em hardware ou em software, as diversas partes do sistema. Finalmente, esta experiência exaltou a necessidade em evitar que os erros cometidos se prolonguem ao longo do processo de desenvolvimento.

2.2 Modelos de processo

Nesta secção, pretende apresentar-se alguns dos modelos de processo mais populares para desenvolvimento de sistemas. Na apresentação é dada maior relevância às propostas feitas na engenharia de software, mas tal facto não tem, segundo o autor, qualquer influência decisiva nos objectivos e nas conclusões desta secção, pois, como se verá na secção 2.5, entre o hardware e o software há muitas semelhanças. Os diversos modelos são descritos de forma sucinta mas suficiente para captar as suas características principais e, para cada um deles, faz-se um pequeno comentário indicando a sua utilização/aplicabilidade nos domínios hardware e software.

Como a fig. 2.1 documenta, a engenharia de software pode ser vista, dum ponto de vista tecnológico, como estando dividida em camadas [Pressman, 1997, pág. 23]. A engenharia de

software, a exemplo dos outros ramos da engenharia, deve fundamentar a sua actividade na qualidade. Tudo o que gravita em torno da engenharia, incluindo obviamente o produto final, deve ser de qualidade, ou, dito de outra forma, esta deve ser o "motor" de todo o trabalho de engenharia. Os alicerces da engenharia de software são os processos. Sob estes assentam os métodos que incluem uma série de tarefas que definem como proceder na análise, especificação, concepção, implementação, teste e manutenção. Por último, os utilitários providenciam um suporte semi-automático ou mesmo automático para o processo e para os métodos. Quando os utilitários estão integradas, no sentido de a informação criada por uma poder ser usada por outra, está-se perante um sistema de apoio ao desenvolvimento de software, chamado *Computer-Aided Software Engineering* (CASE). Um ambiente de CASE para o software é o equivalente dum ambiente de CAD/CAE (*Computer-Aided Design/Engineering*) para o hardware.

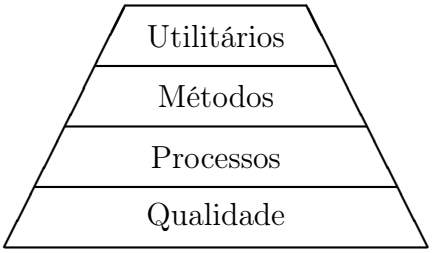


Figura 2.1: As camadas que formam a engenharia de software.

Uma visão alternativa ou complementar a esta indica que as equipas de desenvolvimento, para enfrentarem e resolverem os problemas com que, actualmente, são confrontadas, têm que definir ou adoptar uma metodologia que indique claramente a estratégia a seguir através da definição do processo, dos métodos e dos meta-modelos a usar. A fig. 2.2 pretende ilustrar esta trilogia.

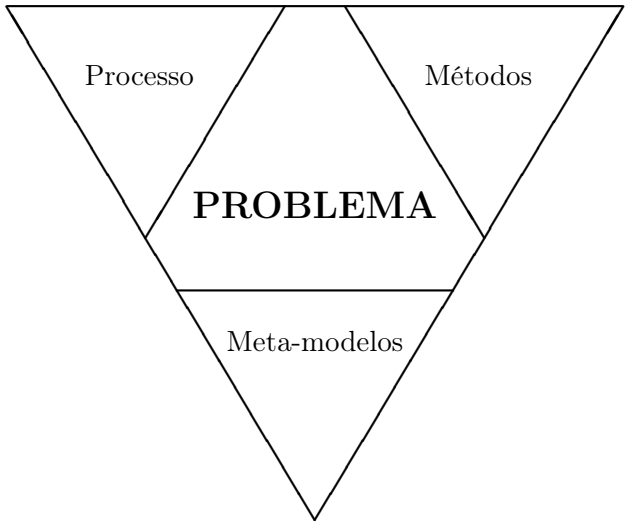


Figura 2.2: Para a resolução dum problema, a metodologia adoptada deve definir o processo, os métodos e os meta-modelos a usar.

Até finais dos anos 70, o desenvolvimento de software era essencialmente uma tarefa realizada (e realizável) por uma só pessoa. O problema a resolver tinha obrigatoriamente de ser simples, pois os computadores tinham recursos muito limitados, em termos de memória e de

velocidade de processamento, pelo que o desenvolvimento consistia unicamente na codificação usando a linguagem de programação escolhida. A produção do software assentava na repetição de dois passos: 1) escrita do código e 2) detecção e eliminação de erros (fig. 2.3). Este modelo de processo, que se designa, em terminologia inglesa, por *code-and-fix*, é totalmente inadequado para sistemas de média ou elevada complexidade, pois apresenta os seguintes problemas [Boehm, 1988]:

1. Após algumas iterações, o código torna-se pouco organizado[1]. Este problema evidencia a necessidade da fase de concepção antes da codificação.
2. É reduzida a relação do código com os requisitos do utilizador, o que demonstra a importância da fase de análise antes da concepção.
3. Torna-se muito dispendioso eliminar os erros do código, evidenciando a relevância de todas as fases de desenvolvimento, bem como da planificação do teste.

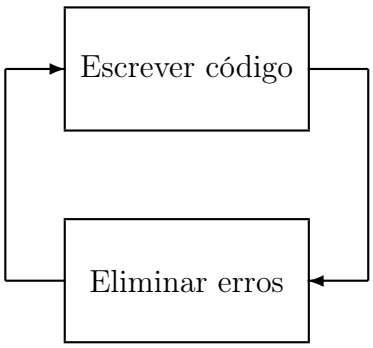


Figura 2.3: O modelo de processo code-and-fix.

Os problemas inerentes ao modelo code-and-fix não emergiram imediatamente, dado que as aplicações eram relativamente simples, o que significa que tanto o domínio da aplicação como o software eram de fácil controlo pelo seu programador (que, normalmente, era também o utilizador final da aplicação).

O falhanço do modelo code-and-fix pode atribuir-se ao facto de, erradamente, se julgar que o software, por ser maleável, não necessita duma abordagem rigorosa para o seu desenvolvimento e que permite que qualquer funcionalidade possa ser conseguida, especialmente quando se tenta alterar algo que já existe. Os profissionais de outros ramos mais tradicionais da engenharia, relativamente aos engenheiros informáticos, são bem mais renitentes em aceitar alterações num dado sistema depois de ele estar implantado e analisam muito mais cuidadosamente o impacto dessas alterações. Esta postura dos engenheiros informáticos faz com que os falhanços na construção de grandes sistemas sejam significativamente bem maiores na engenharia de software que na engenharia civil, por exemplo. Alguns estudos mostraram que por cada 4 projectos de software de larga dimensão que são postos em operação, um deles é cancelado [Gibbs, 1994]. Um número desta grandeza seria simplesmente impensável, hoje em dia, na engenharia civil.

À medida que a complexidade dos problemas foi aumentando, deixou de ser possível que uma única pessoa pudesse desenvolver todo o software. Este facto, aliado ao reconhecimento, por parte da comunidade científica, da inadequação do modelo code-and-fix, para sistema de alguma complexidade, conduziu a um fenómeno designado por *crise do software*, que alguns mais cépticos consideram crónica, mas simultaneamente permitiu, mais tarde, elevar, por direito

[1] A utilização da expressão "código esparguete" pretende vincar o emaranhado em que o código se transforma.

próprio, a engenharia de software à categoria de "verdadeiro" ramo da engenharia[2].

Note-se que, pelo facto de a especificação/modelação dos sistemas hardware ser feita, hoje em dia, com recurso a linguagens de descrição de hardware (HDLs) [De Micheli, 1994, pág. 13], os projectistas deste domínio usam, na maioria das vezes, o modelo *code-and-fix*, sem terem a consciência de que o fazem, dado que não conhecem nenhuma forma alternativa para codificar.

Como corolário do reconhecimento da falta de método na produção do software, surgiu a necessidade em definir e organizar modelos de processo de desenvolvimento, para criar software de qualidade, de modo previsível, sustentado e controlado.

Um *modelo de processo* indica a forma como o processo de desenvolvimento deve ser organizado, para permitir a obtenção do produto final (no caso o sistema), duma forma eficiente, fiável e previsível. Os modelos de processo auxiliam o projectista na organização e relação das actividades e técnicas que compõem o processo de desenvolvimento. Ao definir-se um modelo de processo de desenvolvimento podem colher-se os benefícios da normalização.

A sistematização do processo de desenvolvimento, através da definição de modelos, tem os seguintes objectivos [Yourdon, 1989] [Martin e Odell, 1992]:

- Definir, dum modo claro, as actividades que devem ser prosseguidas para criar os sistemas.
- Introduzir coerência entre os vários sistemas em desenvolvimento. Ao serem definidas actividades padronizadas fica assegurado que todos os sistemas vão ser desenvolvidos da mesma forma.
- Fornecer pontos de controlo (*milestones*) para avaliação dos resultados obtidos e para verificação do cumprimento dos prazos e necessidades de recursos.
- Fomentar uma maior reutilização de componentes, durante as fases de concepção e de implementação dos sistemas, de molde a incrementar a produtividade das equipas de projecto.

De seguida, são referidos e caracterizados alguns dos modelos de processo mais populares que foram propostos com o intuito de eliminar ou, pelo menos, atenuar os problemas que advêm da adopção do modelo code-and-fix.

2.2.1 Modelo em cascata

O mais comum dos modelos de processo de desenvolvimento de software é denominado por *modelo em cascata* (*waterfall model*), sequencial ou linear e é composto, como a fig. 2.4 mostra, por diversas fases, nomeadamente a análise, a concepção, a codificação e o teste [Zave, 1984].

O projectista explicita o funcionamento do sistema, durante a fase de *análise* que inclui as actividades denominadas por *levantamento de requisitos* e *especificação*[3].

Repare-se que o termo especificação pode ser usado para indicar a tarefa em que se especifica, mas também para referir o resultado dessa tarefa, normalmente sob a forma de documento.

[2]Baber considera que a engenharia de software se encontra, actualmente, num estado ainda pouco maduro para ser verdadeiramente considerada um ramo da engenharia, fazendo uma analogia entre o estado actual da engenharia de software e o estado de outros ramos da engenharia (civil, eléctrica, naval) em séculos passados [Baber, 1997]. Numa outra linha de pensamento, Tichy afirma que os profissionais da computação ainda não fazem o número suficiente de experiências para validar algumas das verdades mais antigas da computação, o que reforça a pouca maturidade da engenharia de software [Tichy, 1998].

[3]É também comum designar estas duas actividades por *levantamento dos requisitos do utilizador* e por *levantamento dos requisitos do sistema*, respectivamente, uma vez que, na primeira, a atenção se centra nas necessidades do utilizador do sistema e, na segunda, o esforço é orientado à solução final.

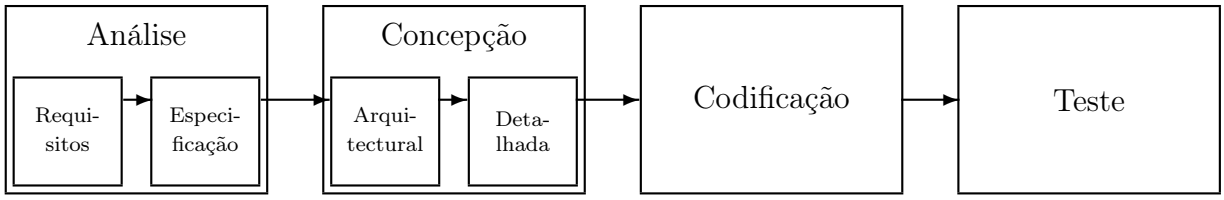


Figura 2.4: O modelo em cascata.

O documento que contém a especificação do sistema servirá de base para as fases seguintes, pelo que, idealmente, a especificação deve definir claramente, sem quaisquer ambiguidades, a funcionalidade pretendida, recorrendo preferencialmente a uma notação independente da forma de implementação, que possa ser entendida por todas as pessoas envolvidas, de alguma forma, no projecto. Na secção 2.4, são abordados, com maior profundidade, diversos tópicos relacionados com a análise de sistemas, uma vez que o contributo mais significativo deste trabalho para a metodologia proposta incidiu precisamente nessa fase de desenvolvimento.

O modelo resultante da análise orientada ao objecto especifica os objectos, as classes e as relações entre estes, mas não indica como é que eles se organizam em estruturas de mais alto nível. É durante a fase de concepção que este problema é resolvido.

Uma vez aceite o documento que especifica o sistema em desenvolvimento, surge, de seguida, a fase de *concepção* que consiste em transformar a especificação numa arquitectura (i.e. atribuir funcionalidades ao hardware e ao software) e que se encontra dividida em duas etapas. A primeira dessas etapas reconhece que uma *arquitectura* do sistema deve ser estabelecida, através da identificação dos módulos em que o sistema se divide e das possíveis restrições no comportamento dos sub-sistemas identificados. Esta arquitectura determina a *estrutura* interna do sistema, definida com base nas entidades que o compõem e nas relações entre essas entidades. A concepção arquitectural descreve como o sistema é constituído e é potencialmente uma das tarefas mais criativas de todo o processo de desenvolvimento [Stevens et al., 1998, pág. 88].

Assim que a arquitectura estiver definida, procede-se então à concepção individual, mais pormenorizada, dos módulos do sistema, de molde a incluir informação suficiente para permitir a codificação do sistema.

Na fase de concepção, o principal objectivo é a estruturação (i.e. a definição duma arquitectura) do sistema em questão. A concepção orientada ao objecto inclui o processo de decomposição orientada ao objecto, usando uma notação apropriada para descrever todos os aspectos (lógicos e físicos, ou estáticos e dinâmicos) relacionados com o sistema [Booch, 1991, pág. 37]. Num sistema orientado ao objecto, a respectiva estrutura é ditada pelos objectos que o compõem e pelas relações que entre eles se estabelecem. O suporte à decomposição orientada ao objecto é o factor que distingue claramente a concepção orientada ao objecto da concepção estruturada: esta recorre a abstracções algorítmicas enquanto que a primeira usa abstracções de classes e objectos.

A diferença fundamental entre as fases de análise e de concepção é que enquanto na primeira se produz um modelo abstracto que mimetiza os aspectos fundamentais do domínio do problema, na última é criado um modelo que especifica os componentes que realizam uma determinada solução desse sistema [Smith e Tockey, 1988]. Resumindo, a fase de análise define qual a funcionalidade do sistema (*o que fazer*), enquanto que a fase de concepção estipula a estrutura (*como fazer*) que o sistema deve apresentar para que o comportamento pretendido seja obtido.

A fase de *implementação* (também conhecida por codificação ou programação) transforma os modelos definidos na fase de concepção em código executável. Essa transformação envolve a definição dos mecanismos internos para que cada módulo satisfaça a sua especificação e a implementação desses mecanismos na linguagem de programação escolhida [Zave, 1984].

A fase de implementação é tida por diversas equipas de investigação [Hatley e Pirbhai, 1988] [Rumbaugh et al., 1991] [Whytock, 1993] como simples e directa, como uma fase meramente mecânica, depois de todo o trabalho mais "pensante" (mais intelectual e mais criativo, se se quiser) ter sido completado nas fases de análise e de concepção. Esta fase é, pois, uma séria candidata a ser automatizada, desde que existam ferramentas que permitam indicar como o código final pode ser gerado a partir das especificações obtidas nas fases anteriores. Contudo, a realidade tem mostrado que nem sempre será assim tão facilitada a persecução desta fase.

A programação orientada ao objecto é um técnica de implementação na qual os programas são organizados como colecções de objectos, cada um dos quais representa uma instância de alguma classe, sendo cada classe um membro duma hierarquia de classes ligadas por relações hierárquicas.

Convém aqui realçar quais as relações entre as fases de análise, concepção e implementação nas metodologias orientadas ao objecto. As especificações criadas na fase de análise servem, como ponto de partida para a fase de concepção. Similarmente, a arquitectura do sistema definida na fase de concepção é usada como documento de referência para implementar o sistema, usando a programação orientada ao objecto [Booch, 1991, pág. 37].

No projecto de sistemas relativamente complexos, as fases de análise e de concepção requerem (ou pelos menos deveriam requerer) uma maior atenção do que a fase de implementação. De facto, a vulgarização de ferramentas CASE (*Computer-Aided Software Engineering*), que produzem automaticamente código a partir de especificações, permite afirmar que se está perto de atingir o ponto em que "*a definição é a implementação*".

A fase de teste é, tradicionalmente, executada no fim do processo de desenvolvimento. Assim que o código estiver totalmente escrito, pode ser executado para, dessa forma, proceder ao seu teste [Kit, 1995, pág. 10]. Contudo, esta visão teve de ser alterada, mal se percebeu que o teste era mais do que uma mera *depuração* do código. O teste de software, se bem conduzido, pode resultar em enormes benefícios económicos, como a seguinte citação esclarece:

> "*The more effective the error detection, the greater the savings in development and maintenance costs over the life of the product. Several examples have indicated that partial testing can yield a saving of 1.5 times its cost; full testing can yield saving of up to 2 times its cost. Whether or not there is universal agreement that these numbers are totally accurate does not matter as much as the fact that they substantiate the premise that testing can invariably pay for itself.*" [Lewis, 1992, pág. 280]

De facto, o sucesso do teste do software depende do planeamento da sua execução e da sua realização efectiva em fases iniciais do desenvolvimento. Se a qualidade do software depende fortemente da qualidade do processo de desenvolvimento adoptado, de modo semelhante, a qualidade e a eficácia do teste são largamente determinadas pela qualidade do processo de teste usado [Kit, 1995, pág. 3]. Actualmente, o *teste* de software tem o seu próprio ciclo que é realizado a vários níveis: inicia-se ao mesmo tempo que o levantamento de requisitos, e a partir daí, segue em paralelo com o próprio processo de desenvolvimento. Por outras palavras, para cada fase ou actividade do processo de desenvolvimento existe associada uma actividade de

teste, como a fig. 2.5 ilustra [Robinson, 1992, pág. 3]. Repare-se que esta figura, que representa o *modelo em V*, pode ser vista como um refinamento do modelo em cascata apresentado na fig. 2.4.

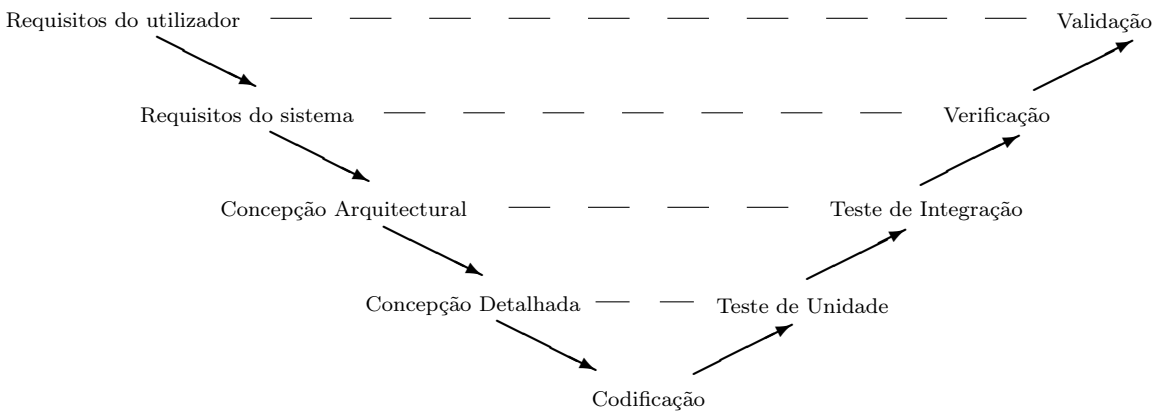


Figura 2.5: O modelo em V.

O primeiro nível de teste (*teste de unidade*) realiza-se, à medida que os diversos módulos (unidades) vão sendo codificados, usando os resultados obtidos na concepção detalhada como termo de comparação. O propósito do *teste de integração* consiste em mostrar que o software, como um todo, é coerente com a concepção arquitectural. A ênfase é habitualmente colocada no teste das interfaces entre componentes. Note-se que os testes aos modelos da concepção podem revelar-se actividades muito exigentes em termos de recursos e tempo, especialmente para sistemas críticos em termos de segurança ou fiabilidade. A *verificação* permite testar se o software satisfaz totalmente todos os requisitos indicados na especificação do sistema. Finalmente, a *validação* é realizada pelo utilizador final que compara o funcionamento do sistema relativamente às suas próprias expectativas. Todas estas operações de teste, com excepção da validação, podem ser automatizadas, parcial ou mesmo totalmente, dependendo essa possibilidade das notações usadas nas fases de análise e concepção.

Todas estas fases (análise, concepção, codificação e teste) estão relacionadas entre si e nenhuma delas deve ser negligenciada, durante o desenvolvimento dum determinado sistema. A maior parte das vezes, a ordem em que são executadas é difícil de precisar, não sendo necessário que uma fase termine totalmente para que a seguinte possa ser iniciada. Habitualmente, os resultados que vão sendo obtidos numa dada fase são usados para corrigir os resultados de fases anteriores. Na prática, é desejável que qualquer processo seja iterativo, dado que problemas detectados em fases mais avançadas do projecto obrigam, normalmente, a revisitar fases anteriores.

Todos os projectos incluem, de alguma forma, as fases de análise, concepção, e codificação, mas a divisão entre estas fases nem sempre é tão explícita como até agora foi indicado. Por exemplo, nas metodologias orientadas ao objecto, há, normalmente, uma sobreposição nas tarefas abrangidas nas fases de análise e de concepção, pelo que a separação entre essas duas fases é mais conceptual do que real, uma vez que a respectiva fronteira é pouco nítida. A seguinte citação corrobora esta ideia:

"*The boundaries between analysis and design are fuzzy.*" [Booch, 1994, pág. 155].

Apesar deste facto poder ser interpretado como uma desvantagem, uma leitura positiva também é possível, pois tal significa que a transição entre as fases se faz duma forma mais natural, quando se segue uma decomposição orientada ao objecto.

Por outro lado, é possível organizar as diversas fases de desenvolvimento de forma ligeiramente diferente à que se apresenta na fig. 2.4. É aceitável que as fases em que se realiza o estudo de viabilidade e a análise do sistema sejam agrupadas numa só. Também é possível que as duas sub-fases de concepção (arquitectural e detalhada) sejam reunidas numa única fase de concepção. Similarmente, as várias actividades de teste são frequentemente incluídas em apenas uma fase. Contudo, todas estas variantes não são, na sua essência, distintas do modelo em cascata apresentado na fig. 2.4, já que a característica mais relevante deste modelo é a forte tendência para um desenvolvimento segundo uma abordagem descendente (do mais abstracto até ao concreto) e a progressão sequencial ou linear entre fases consecutivas [Yourdon, 1988, pág. 45–7].

Comentários

O modelo em cascata, devido à sua simplicidade conceptual, que permite relacionar com facilidade as várias fases, continua a ser o processo mais usado tanto no desenvolvimento de software como no desenvolvimento de hardware, apesar dos vários problemas que apresenta e dos diversos modelos alternativos que foram propostos para guiar o projectista no desenvolvimento de sistemas. O modelo em cascata é considerado muito rígido e produz resultados satisfatórios somente quando os requisitos são claros e pouco susceptíveis de se alterarem de forma significativa. Para desenvolver um compilador, por exemplo, em que a gramática da linguagem está completamente definida e não é previsível que seja necessário alterá-la, este modelo parece adequar-se perfeitamente [Ghezzi et al., 1991, pág. 374].

No entanto, o modelo em cascata, por se basear em documentos como critério para determinar a finalização duma fase e a passagem entre fases, obriga a escrever documentos completos para que se possa transitar para a fase seguinte. Este facto revela-se extremamente negativo, em projectos onde os requisitos não são totalmente conhecidos, por obrigar a escrever especificações completas (e muito provavelmente com erros), a que se seguem depois as fases de concepção e de implementação que irão herdar esses erros. Infelizmente, só quando a implementação estiver finalizada é que esses erros poderão ser detectados.

Apesar da fase de análise ser fundamental para o bom sucesso no desenvolvimento de qualquer sistema, seja ele de software ou de hardware, a prática tem mostrado que, contrariamente ao que seria desejável, no desenvolvimento de sistemas hardware esta fase é muita vezes esquecida ou então realizada de forma pouco metódica. O levantamento dos requisitos é, muitas vezes, neglicenciado, devido, entre outras, às seguintes razões [Zave, 1982]:

- Falta de percepção da sua importância, no contexto do desenvolvimento do sistema.
- Desconhecimento de técnicas para análise e especificação.
- Relutância em assumir custos e atrasos no início do projecto.

Grande parte dos modelos de processo, para sistemas hardware, apresenta a especificação do sistema como primeira entrada para todo o processo de desenvolvimento, assumindo a sua correcção, ou melhor, não pondo em questão o processo utilizado para obter essa especificação. Exemplos concretos desta realidade são os modelos de processo apresentados pelas seguintes equipas de investigação: [De Micheli, 1994] [Thomas et al., 1993] [Swamy et al., 1995] [Camposano e Wilberg, 1996] [Kleinjohann et al., 1997] [Machado et al., 1997b].

2.2.2 Modelo iterativo

O cliente, por vezes, define apenas alguns objectivos genéricos para o sistema, não indicando, com pormenor, as entradas esperadas, o processamento a realizar, ou o tipo das saídas. Numa situação idêntica a esta, uma abordagem baseada na *prototipagem* pode ser a escolha mais acertada. Esta abordagem tem associado um *modelo de desenvolvimento iterativo* (que pode também designar-se por modelo evolutivo), cuja estrutura está representada na fig. 2.6.

A prototipagem inicia-se com o levantamento dos requisitos; nesta tarefa, o cliente e o projectista reúnem-se e definem os requisitos conhecidos e identificam áreas em que será necessário intervir futuramente, através do refinamento dos requisitos anteriormente considerados e da definição de novos. De seguida, uma construção rápida do sistema ocorre, significando aqui o termo "rápida" o facto de serem desenvolvidas versões rudimentares e temporárias do sistema, vulgo *protótipos* [Boar, 1984], em que apenas são considerados alguns dos aspectos do sistema, nomeadamente aqueles que dizem respeito à vista externa do sistema (a relação entre as entradas e as saídas).

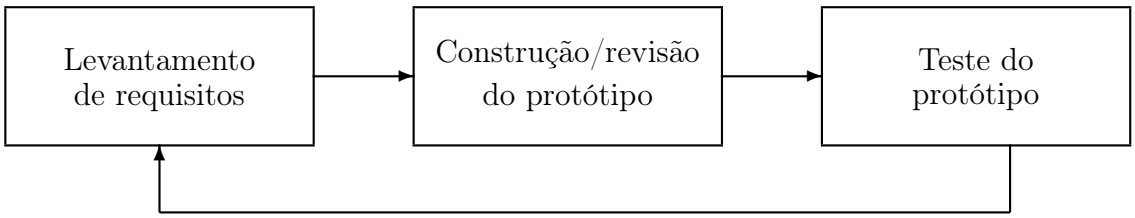


Figura 2.6: O modelo de processo iterativo, seguido na prototipagem de sistemas.

O protótipo construído é avaliado pelo cliente e é usado para refinar os requisitos do sistema. O carácter evolutivo da prototipagem deve-se ao facto de os protótipos serem gradualmente alterados para satisfazerem, cada vez mais, as necessidades do cliente, com o objectivo de permitir que, em cada iteração, o projectista compreenda melhor o sistema que necessita de ser desenvolvido.

Teoricamente, o protótipo serve simplesmente como mecanismo para captar os requisitos do sistema. Logo que se considera que os requisitos do cliente estão claramente entendidos, usualmente o protótipo do sistema é abandonado[4] e começa-se então o processo de desenvolvimento, tendo em atenção os resultados obtidos com a ajuda do protótipo e seguindo, por exemplo, um processo de desenvolvimento segundo o modelo em cascata.

Uma das vantagens principais da prototipagem reside na possibilidade em se iniciar o desenvolvimento do sistema, mesmo não conhecendo totalmente os requisitos pretendidos para o sistema. Adicionalmente, possibilita ao projectista começar, muito cedo, a construir algo que realmente funciona, o que permite que os utilizadores tenham uma ideia quanto à forma como o sistema se irá comportar, o que não sucederia tão facilmente caso o utilizador fosse confrontado com uma descrição do sistema em papel.

Como desvantagens, este modelo apresenta o facto de tanto o cliente como o projectista se esquecerem, muitas vezes, que o protótipo não é uma versão final do sistema, mas antes um meio para facilitar a captura dos requisitos. O protótipo foi desenvolvido, assumindo uma série de factos (escolha duma linguagem ou dum sistema operativo ineficientes, dum algoritmo

[4]Há outras perspectivas da prototipagem, mas, por questões de simplicidade, assume-se apenas esta.

pouco apropriado ou com desempenho inaceitável), de forma a simplificar o processo de desenvolvimento, a fim de captar unicamente os requisitos do sistema. Contudo, depois de verem o protótipo a funcionar, o cliente e o projectista são naturalmente impelidos pela tentação de acharem que umas ligeiras alterações no protótipo podem permitir obter uma versão do sistema com níveis de desempenho razoáveis, o que raramente corresponde à verdade.

Comentários

Este modelo é nitidamente oriundo do domínio hardware, onde existe uma enorme tradição na construção de protótipos (ou de versões *bread boarded*). Os protótipos são usados principalmente para teste da solução e raramente para validação dos requisitos do utilizador. Esta técnica foi, posteriormente, adaptada para desenvolver software, havendo contudo, neste domínio, um maior hábito em utilizar a prototipagem para captar os requisitos do utilizador.

2.2.3 Modelo incremental

O *modelo incremental* combina as características do modelo em cascata (aplicado repetidamente) com a filosofia iterativa subjacente à prototipagem. Como a fig. 2.7 ilustra, este modelo de processo aplica sequências lineares de desenvolvimento duma forma faseada. Cada sequência linear produz como resultado um incremento funcional do software final. Por exemplo, a construção dum editor de texto poderia ser realizada, seguindo as seguintes iterações:

- Na 1^a iteração, desenvolvem-se as funções para manipulação de ficheiros (abrir, gravar, fechar, imprimir) e funcionalidades básicas para edição de texto (inserir, apagar, seleccionar).
- Na 2^a iteração, incluem-se as capacidades de edição mais avançadas (procurar e substituir texto, tipos de letra, negrito, sublinhado).
- Na 3^a iteração, a inclusão de figuras, tabelas e gráficos é facilitada, através da disponibilização de novos comandos.
- Na 4^a iteração, acrescentam-se funcionalidades que permitam o uso de correctores automáticos e de dicionários de sinónimos.

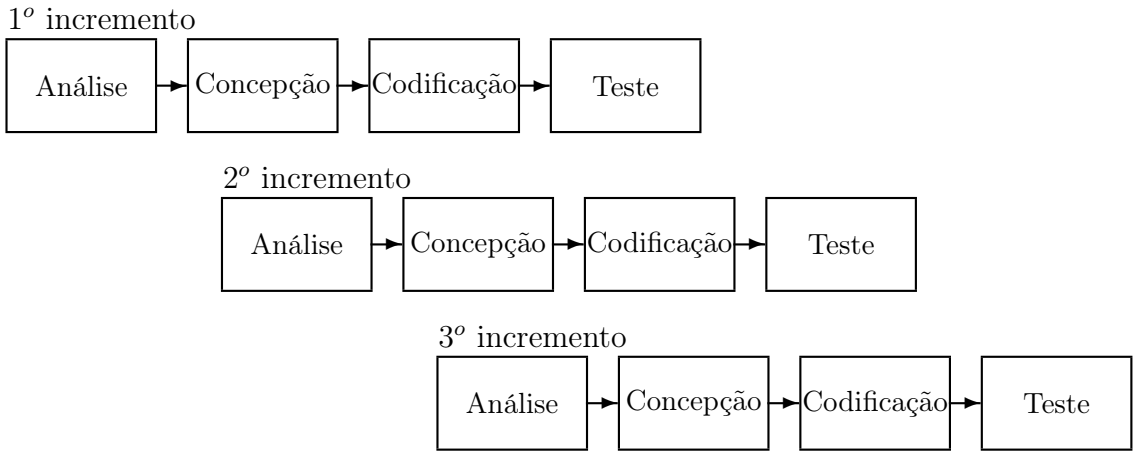


Figura 2.7: O modelo incremental.

A modelação incremental baseia-se na ideia de que é mais fácil criar uma estrutura simples do que uma outra complexa e de que também é mais simples modificar uma estrutura já existente do que criar uma nova de raiz (a partir do nada). Assim, em vez de tentar criar completamente o modelo num só passo, as atenções, no início, concentram-se deliberadamente na incorporação dos aspectos mais críticos e importantes (ou aqueles mais claramente captados), sendo depois o modelo repetidamente melhorado, até estar completo.

O modelo incremental, a exemplo do que acontece com a prototipagem, é, na sua essência, repetitivo e iterativo, i.e. as funcionalidades vão sendo acrescentadas até o sistema estar completamente descrito. No entanto, contrariamente ao que se verifica na prototipagem, o modelo incremental centra a sua acção na construção do produto final. Cada incremento representa uma versão reduzida do sistema final, que terá um contributo para a funcionalidade do sistema.

Comentários

Este modelo é usado em ambos os domínios (software e hardware) e depende muito mais da complexidade inerente ao sistema do que propriamente da forma que se escolheu para implementar o sistema. Trata-se, portanto, duma técnica que, aliada a outras, permite atacar a complexidade dos sistemas.

2.2.4 Modelo transformacional

Apesar de ainda não serem, hoje em dia, uma solução generalizada para todo o tipo de sistemas, os métodos matemáticos (habitualmente conhecidos por métodos formais) oferecem a perspectiva muito atraente de supostamente gerarem software sem quaisquer erros. Contudo, entre as desvantagens da sua utilização, encontram-se o longo tempo de desenvolvimento, a dificuldade em usar as especificações como meio de comunicação com os clientes e a necessidade de altos especialistas para manipularem as especificações de carácter matemático[5]. Além disso, o sucesso na aplicação de métodos formais depende fortemente do cumprimento de algumas regras, que infelizmente nem sempre são seguidas [Bowen e Hinchey, 1995].

O uso de métodos formais pressupõe um *modelo transformacional* que assume a existência de ferramentas que automaticamente convertem uma especificação formal num programa de software que satisfaz aquela especificação. Este modelo elimina assim as dificuldades que se observam na modificação do código que se torna pouco estruturado através de optimizações repetidas, uma vez que as alterações são feitas na especificação [Boehm, 1988].

A fig. 2.8 ilustra o modelo transformacional que é, idealmente, composto por 2 etapas principais: análise de requisitos e optimização. A primeira etapa produz como resultado uma especificação formal que será a entrada da etapa seguinte. Nesta, a especificação vai sendo "afinada", até se obter uma solução satisfatória. O processo é controlado pelo engenheiro de software, sendo possível a utilização de componentes reutilizáveis, com as conhecidas vantagens que daí resultam. Como a figura também mostra, durante o processo, novos componentes reutilizáveis podem ser desenvolvidos e armazenados em bibliotecas para utilização futura. Antes de ser transformada, a especificação formal é verificada relativamente às necessidades do cliente, podendo utilizar-se, entre outras técnicas, provas formais de propriedades ou execução de software.

[5]Este argumento não deve, contudo, ser tomado literalmente, pois alguns métodos formais usam conceitos familiares a qualquer engenheiro. Por exemplo, o método Camila [Oliveira, 1995] usa, para especificar os sistemas software, a teoria dos conjuntos, dada habitualmente, em Portugal, nos ensinos preparatório e secundário.

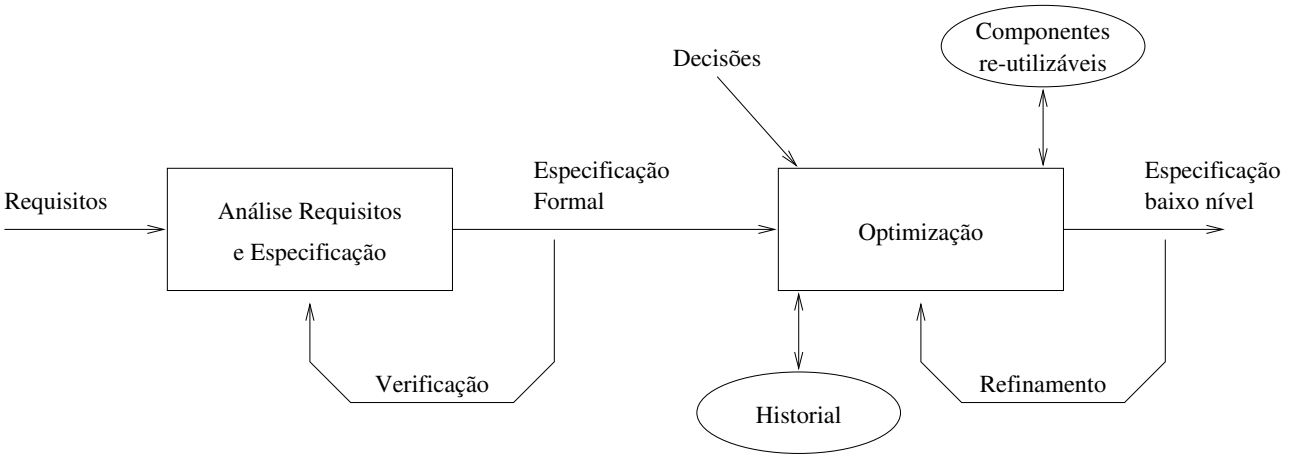


Figura 2.8: O modelo transformacional.

Este modelo de processo torna-se eficiente e útil, se existir um ambiente que disponibilize ferramentas para suporte automático às várias actividades (verificação dos requisitos, manipulação de componentes reutilizáveis, optimizações, geração de historial do processo).

No modelo em cascata, as alterações são entendidas como reparações, uma vez que a sua ocorrência não é antecipadamente considerada. Daqui resulta que as alterações são normalmente feitas sob grande pressão, normalmente no fim do desenvolvimento, pelo que, quase sempre, são realizadas modificando-se directamente o código, sem fazer reflectir essas alterações na especificação. Assim, a especificação e a implementação vão rapidamente divergindo uma da outra, tornando eventuais alterações ainda mais difíceis de realizar no futuro. Esta situação não se verifica no modelo transformacional, visto que a história do desenvolvimento do software e as respectivas decisões (passos intermédios, baseados em provas matemáticas) para cada transformação estão gravadas, o que possibilita ao programador recomeçar, a partir dum ponto intermédio, a transformação da especificação numa implementação.

Comentários

Este modelo está, por tradição, intrinsecamente associado ao software, onde, relativamente ao hardware, a utilização de métodos formais está melhor sustentada. Contudo, nos últimos tempos, tem-se assistido a um crescente interesse na utilização dos métodos formais e consequentemente do modelo transformacional, no domínio hardware [Delgado Kloos e Breuer, 1995] [Barringer et al., 1997].

Apesar da acrescida maturidade que os métodos formais têm vindo a adquirir, a sua utilização ainda não se disseminou de forma global, estando a sua aplicação restringida a algumas áreas bem delimitadas.

2.2.5 Modelo em espiral

O *modelo em espiral* [Boehm, 1988] tem por objectivo disponibilizar uma plataforma para conceber modelos de processo e baseia-se numa abordagem orientada ao risco e não baseada

em documentos ou no código, como, por exemplo, os modelos em cascata ou code-and-fix, respectivamente. Neste contexto, *riscos* são circunstâncias potencialmente adversas que podem ter efeitos negativos no processo de desenvolvimento e na qualidade final do sistema. O modelo em espiral centra a sua acção na identificação e eliminação de problemas com alto risco.

Este modelo, por incluir os modelos anteriores como casos particulares, facilita a escolha da melhor combinação desses modelos para cada uma das situações em que é aplicado, pelo que pode ser visto como um meta-modelo (i.e. um modelo para criar modelos de processo).

Neste modelo, as várias fases são organizadas por ciclos, como a fig. 2.9 documenta. Cada ciclo da espiral é constituído por 4 tarefas principais, sendo cada uma delas representada por um quadrante do diagrama. O raio da espiral representa o custo acumulado no processo e a dimensão angular indica o progresso no processo.

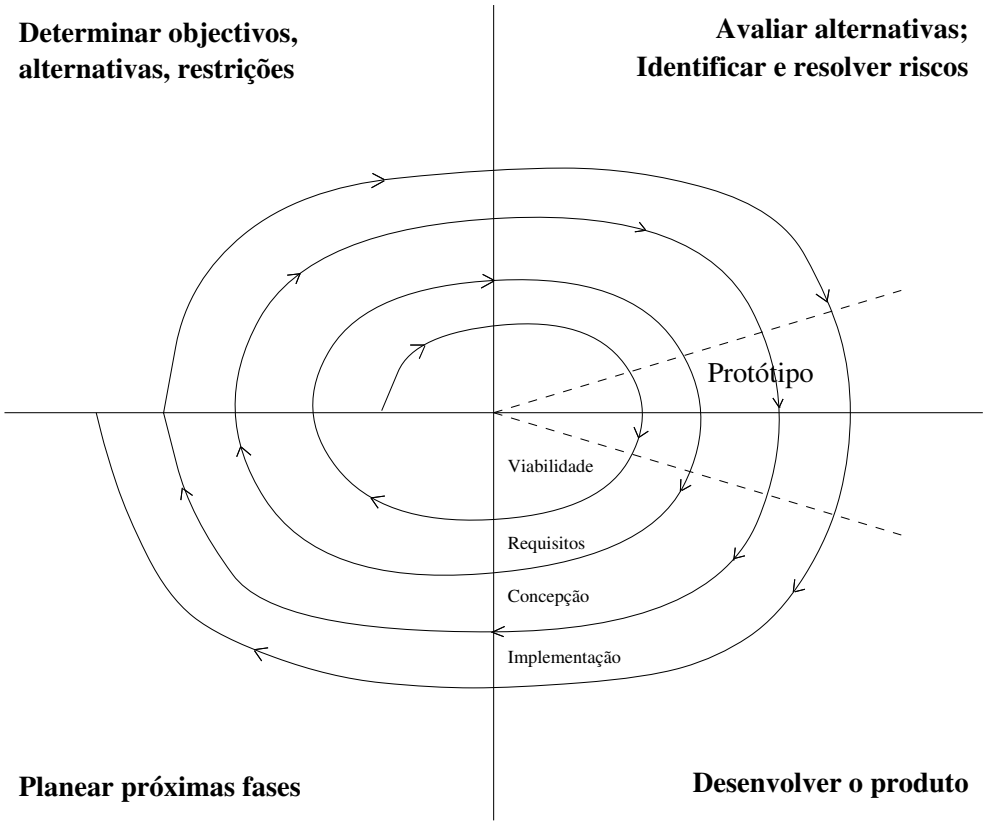


Figura 2.9: O modelo em espiral.

Na primeira tarefa do ciclo, identificam-se os objectivos (desempenho, funcionalidade, facilidade de modificação, etc.) para o sistema em estudo, relativamente aos níveis de qualidade a atingir. São ainda identificados os meios alternativos para implementação (desenvolver A ou B, comprar, reutilizar, etc.) e as restrições que se têm de impor na aplicação dessas alternativas. Na segunda fase do ciclo, procede-se à avaliação das alternativas anteriormente identificadas em relação aos objectivos e restrições, o que frequentemente implica a identificação de situações de incerteza que representam potenciais fontes de risco. Para proceder a esta identificação pode recorrer-se a várias técnicas, como prototipagem, *benchmarking*, simulação ou questionários. Durante a terceira fase, desenvolve-se e verifica-se o sistema para o próximo ciclo, baseando-se novamente numa estratégia orientado ao risco. Na quarta e última fase do ciclo, revêem-se os resultados

das fases anteriores e planeia-se o próximo ciclo da espiral (se eventualmente for caso disso).

Quando os requisitos da aplicação são razoavelmente bem conhecidos, um processo segundo o modelo em cascata pode ser seguido, o que significa que apenas um ciclo da espiral é cumprido. Para sistemas cujos requisitos sejam menos claros, podem ser necessários vários ciclos para alcançar os resultados desejados, de que resulta um processo iterativo. Estas duas situações são devidamente descritas pela fig. 2.10 que apresenta uma nova disposição do modelo em espiral, atribuindo uma das fases do desenvolvimento a cada fatia do diagrama.

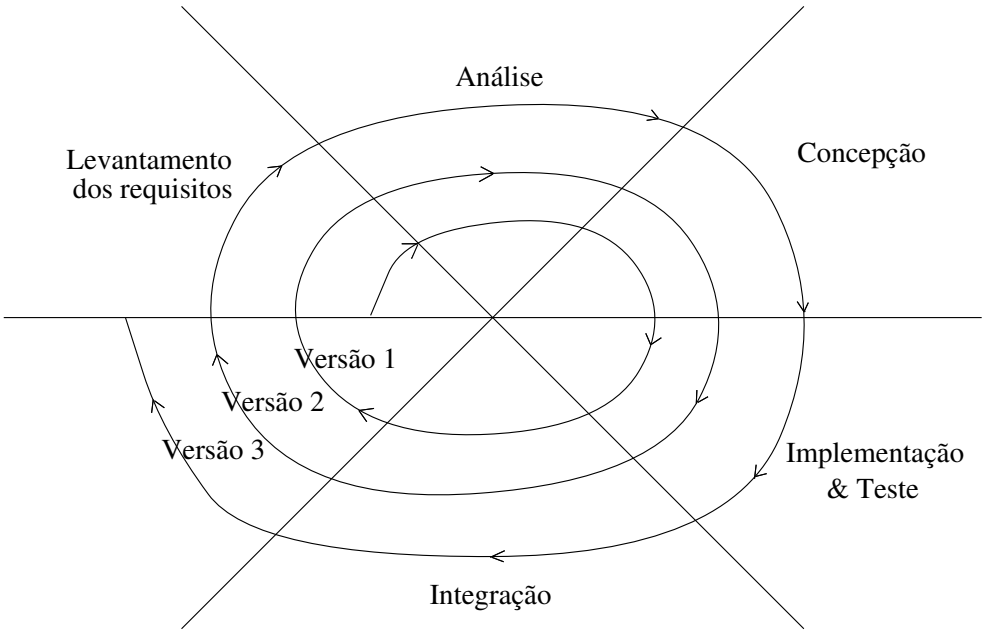


Figura 2.10: O modelo em espiral, segundo uma nova disposição.

Comentários

Este modelo foi introduzido inicialmente no software, mas a sua aplicabilidade a sistemas hardware é perfeitamente possível, uma vez que os riscos associados ao desenvolvimento dum sistema não dependem da forma como ele se implementa.

2.3 Metodologias de desenvolvimento

O conjunto de reflexões feitas na secção anterior, bem como aquele que será elaborado na secção 2.5, permite concluir que o desenvolvimento de sistemas de software e de hardware, à parte de pequenas particularidades, é, em grande parte, semelhante. Este pressuposto permite que, nesta secção, se concentre o estudo na forma como o desenvolvimento de software foi historicamente evoluindo e que, desse estudo, se possam tirar algumas conclusões sobre o modo como se podem e devem desenvolver sistemas embebidos.

2.3.1 Metodologias estruturadas

Nos anos setenta, surgiram os primeiros esforços "sérios" na definição de metodologias para a concepção de sistemas de software de grande dimensão: as *metodologias estruturadas*. Como resultado, no final dessa década, alguns princípios eram tidos como amplamente aceites, nomeadamente, o uso de diagramas como forma principal de documentação e a importância de mecanismos hierárquicos e de abstracção para permitir que os projectos fossem conduzidos duma forma mais clara e simples. Alguns outros tópicos foram também reconhecidos como importantes: a necessidade de decidir primeiro qual a funcionalidade dos sistemas, antes de explicitar como o conseguir; as vantagens da filosofia "dividir para reinar", relativamente fácil de aplicar se existirem mecanismos poderosos de modularização; e a utilidade indiscutível do encapsulamento (subsecção 3.2.4).

As metodologias estruturadas [DeMarco, 1979] [Gane e Sarson, 1979] [Yourdon, 1989], e as extensões propostas para sistemas de tempo-real [Ward e Mellor, 1985] [Hatley e Pirbhai, 1988], focam essencialmente a modelação do fluxo de dados do sistema.

As metodologias estruturadas têm a sua origem na arquitectura proposta por von Neumann, onde existe uma acentuada separação entre o programa e a estrutura de dados. De facto, nos sistemas desenvolvidos segundo a abordagem estruturada, constata-se uma divisão muito nítida entre a funcionalidade e os dados que compõem esse sistema. Uma metodologia estruturada recorre, na fase de análise, fundamentalmente, ao *diagrama de fluxo de dados* (DFD) para descrever o sistema e a decomposição deste é feita com base na sua funcionalidade. Um sistema é, pois, visto como um fornecedor de funções para o utilizador. O desenvolvimento é iniciado com um único processo (ou função), que representa a funcionalidade total do sistema em causa. O uso de diagramas de contexto, em que o sistema é representado como uma entidade única ligada aos elementos com que comunica no seu ambiente, é usado exactamente como o primeiro DFD do sistema em algumas metodologias [Hatley e Pirbhai, 1988] [Morris et al., 1996].

De seguida, o processo inicial é dividido numa série de outros processos menos complexos e esta divisão vai sendo aplicada, repetidamente, até sobrarem apenas funções de "fácil" implementação. Os sistemas são sujeitos a uma *decomposição funcional*, que é definida como o processo de desenvolver uma implementação para uma função, expressa como um algoritmo que conjuga uma colecção de funções mais simples. A decomposição funcional é uma técnica de refinamento que utiliza abstracções funcionais para gerir a complexidade de forma estruturada, seguindo uma abordagem descendente (i.e. do mais complexo para o mais simples).

Foram propostas diversas metodologias estruturadas (além das supramencionadas, refira-se ainda [Myers, 1978] [Yourdon e Constantine, 1979] [Page-Jones, 1988]), mas nenhuma se conseguiu impor como a "*oficial*", apesar das enormes pressões governamentais que se exerceram nesse sentido. O motivo para esse insucesso tem explicação no facto de ser muito complicado definir uma metodologia que abarque todos os tipos de aplicações e que agrade a todos os seus (da metodologia) potenciais utilizadores. Por exemplo, uma metodologia adequada para aplicações de processamento de dados, não será certamente a mais indicada para aplicações de controlo automático de processos. Este motivo levou ao aparecimento de inúmeras metodologias, cada uma tentando ocupar o seu nicho de mercado, a sua área de influência ou o seu domínio de aplicação.

Um dos problemas mais acentuados das metodologias estruturadas deriva do facto de a análise inicial se basear numa decomposição funcional do sistema, da qual resulta uma série de funções que partilham o mesmo conjunto de dados. Esta estrutura é, nos tempos actuais, considerada

altamente inadequada. Genericamente, os meta-modelos usados nas metodologias estruturadas apresentam várias limitações importantes, entre as quais merecem realce [Booch, 1986]:

- Não incorporam, de forma simples, abstracção de dados (subsecção 3.2.3), nem encapsulamento (subsecção 3.2.4).
- São inadequados para problemas com concorrência.
- Tornam-se instáveis a alterações que surgem, inevitavelmente, ao longo de qualquer problema minimamente complexo.

2.3.2 Metodologias orientadas ao objecto

As limitações presentes nas metodologias estruturadas levaram ao aparecimento das *metodologias orientadas ao objecto*, que, duma forma simplista, são abordagens para a concepção de software, nas quais a decomposição do sistema é baseada no conceito de objecto (dados do sistema). Esta abordagem contrasta com as metodologias estruturadas, em que o sistema sofre uma decomposição do tipo funcional ou algorítmica (i.e. baseada nas funcionalidades do sistema). Alguns autores têm mesmo uma visão mais evolutiva dos dois tipos de metodologias e consideram que as técnicas do paradigma dos objectos constituem uma extensão àquelas que são disponibilizadas pela abordagem estruturada [Tockey et al., 1990].

O termo "orientado ao objecto", aplicado a um sistema, significa que este é organizado como uma colecção de objectos que incorporam, simultaneamente, estrutura e comportamento. Esta perspectiva diverge da programação mais tradicional, em que as entidades (estruturas de dados) e o comportamento (funções e procedimentos) estão apenas tenuemente ligados.

A essência da modelação orientada ao objecto é a identificação e organização de conceitos do domínio de aplicação, em vez da sua representação final numa dada linguagem de programação. Numa abordagem orientada ao objecto, o sistema é dividido, segundo uma *decomposição orientada ao objecto*, tomando em consideração os objectos, ou seja, as coisas, que realmente existem no domínio do problema. As vantagens desta decomposição são consideráveis, visto que as alterações na representação dum objecto são mais localizadas. Numa decomposição funcional, dado que as estruturas de dados importantes são, necessariamente, de âmbito global, uma alteração, por exemplo, no tipo de dados duma variável obriga a alterar todas as funções que dela fazem uso.

Pode, com todo o sentido, colocar-se a questão de qual a forma mais correcta para decompor um sistema complexo: por algoritmos ou por objectos? Infelizmente, parece não haver uma resposta inequívoca para esta pergunta. Não é credível dizer, sem qualquer dúvida, que raciocinar em termos de objectos é mais natural do que raciocinar em termos de funções ou eventos. Ambas as abordagens são importantes: a perspectiva algorítmica realça a ordem dos eventos, ao passo que a perspectiva orientada ao objecto se centra nas entidades que causam acções ou que são sujeitas a ser actuadas por operações.

Não é possível construir, simultaneamente, um sistema complexo dos dois modos. Tem obrigatoriamente que se começar a decomposição do sistema pelos algoritmos ou pelos objectos e, depois, pode usar-se a estrutura resultante como plataforma para descrever a outra perspectiva. É, hoje, comummente aceite que a decomposição do sistema, seguindo, em primeiro lugar, uma abordagem orientada ao objecto, facilita a organização da complexidade inerente aos sistemas.

A decomposição orientada ao objecto apresenta um conjunto muito significativo de vantagens relativamente à decomposição algorítmica. Daquela resultam sistemas mais pequenos através

da reutilização de mecanismos comuns, possibilitando uma economia no esforço de expressar os sistemas. Os sistemas orientados ao objecto são também conhecidos por se adaptarem mais rapidamente a alterações, podendo assim evoluir facilmente no tempo, uma vez que a sua modelação se baseia em formatos mais estáveis [Booch, 1991, pág. 71].

O desenvolvimento de sistemas, seguindo uma abordagem orientada ao objecto, ou mais simplesmente o *desenvolvimento orientado ao objecto*, é definido como a utilização consciente e deliberada de objectos como critério de organização para as abstracções de dados e os procedimentos do sistema [Tockey et al., 1990].

Para finalizar, refira-se que, a exemplo do que sucedeu com as metodologias estruturadas, também para as metodologias orientadas ao objecto não foi ainda possível estabelecer oficialmente (ou apenas de facto) uma delas como norma, por motivos idênticos aos indicados para as primeiras (pág. 35). As inúmeras metodologias de desenvolvimento orientado ao objecto, bem como os métodos de análise e concepção, propostos nos últimos anos[6], suscitam ao autor duas leituras distintas. Por um lado, trata-se dum factor positivo, pois significa que inúmeras equipas de investigação estão a tratar, da mesma forma (paradigma dos objectos), o mesmo problema (desenvolvimento de software), embora com abordagens distintas. Por outro lado, pode fazer-se uma leitura negativa, pelo facto de se ter instalado um clima de competição entre os "vendedores" de metodologias, o que pode levar algumas equipas de desenvolvimento a perder muito tempo para decidir qual é a vencedora da *guerra das metodologias* (ou seja, na escolha de qual a "melhor" metodologia para as suas peculiaridades).

Num painel duma conferência [Monarchi, 1994], a divergência de opiniões acerca da validade e da utilidade duma metodologia oficial foi evidente, o que permitia antever a dificuldade na tentativa de estabelecer uma metodologia padrão para o desenvolvimento orientado ao objecto.

Este cenário pessimista começou a alterar-se (para melhor), quando em Novembro de 1997, a notação UML (secção 3.3) foi instituída como a norma da OMG. Anteriormente, todas as metodologias apresentavam a sua própria notação, que tinham, pormenores à parte, um conjunto muito significativo de características comuns. Apesar de se tratar apenas duma notação (independente dos métodos e dos processos), UML pode vir a revelar-se como o trunfo final para popularizar, principalmente em meios industriais, as técnicas orientadas ao objecto. De facto, aquilo que é usado como meio para veicular a informação é a especificação e não a metodologia, pelo que, havendo uma notação normalizada, todos os agentes envolvidos no desenvolvimento de sistemas passam a "falar" a mesma linguagem. Por exemplo, um consultor passará a ter a sua actividade científica mais facilitada, uma vez que não precisa de se inteirar da notação, antes de ser capaz de ler e interpretar a especificação.

O aparecimento de *metodologias de segunda geração* (i.e. metodologias que tentam integrar as características mais importantes e positivas de outras metodologias, mas com o cuidado de criar um corpo coerente[7]) também pode ajudar a tornar o panorama menos sombrio. As metodologias de segunda geração, de que são exemplos paradigmáticos Booch'94 [Booch, 1994] e *Fusion Method* [Coleman et al., 1994], podem vir a revelar-se muito úteis, sobretudo para uma organização que pretenda iniciar-se no desenvolvimento orientado ao objecto, porque atenua a

[6] [Hutt, 1994] apresenta uma compilação de 21 métodos de análise e concepção que seguem o paradigma dos objectos e [Sigfried, 1996, pág. 430–6] apresenta, na bibliografia, uma extensa lista de referências sobre metodologias orientadas ao objecto.

[7] Deve ter-se o cuidado de evitar aqui o síndroma PL/I [ANSI, 1976], uma linguagem criada para incluir uma série de mecanismos disponíveis em Cobol, Fortran e outras linguagens procedimentais, mas que acabou por se revelar um fracasso, por não constituir um todo harmonioso.

probabilidade de essa organização se envolver na guerra das metodologias.

2.3.3 Comparação entre as abordagens estruturada e orientada ao objecto

Não existe, infelizmente, nenhuma "varinha mágica"[8] que faça com que o desenvolvimento de sistemas complexos seja uma tarefa trivial. Neste sentido, far-se-á uma confrontação entre as abordagens estruturada e orientada ao objecto, tentando mostrar que a última é "melhor" que a primeira, não porque torna mais simples o processo de desenvolvimento de sistemas, mas porque disponibiliza uma série de técnicas e mecanismos que permitem desenvolver, mais rapidamente e com menos erros, sistemas complexos. Uma vez que, nas secções 2.3.1 e 2.3.2, foram já esgrimidas algumas reflexões relativas a esta comparação, o que a seguir se apresenta será um complemento a essas reflexões.

As metodologias estruturadas não abordam, de forma apropriada, os aspectos dinâmicos dos sistemas reactivos, uma vez que este tipo de metodologia foi proposto, principalmente, para lidar com aplicações transformacionais (orientadas aos dados e não reactivas), para as quais uma adequada decomposição funcional e uma boa descrição baseada no fluxo dos dados são suficientes [Harel et al., 1990]. As extensões para sistemas de tempo-real [Ward e Mellor, 1985] [Hatley e Pirbhai, 1988], que foram surgindo, ao longo dos tempos, para as metodologias estruturadas, vieram claramente confirmar a inadequação destas para o desenvolvimento de sistemas reactivos.

A análise orientada ao objecto, relativamente à análise estruturada, é tida como mais natural, para modelação dum sistema, visto que os objectos correspondem a entidades do domínio da aplicação. Por outro lado, a passagem da fase de análise para a fase de concepção, embora não sendo um processo directo e imediato, é também considerada como mais natural quando se usam metodologias orientadas ao objecto, uma vez que há um refinamento e uma evolução dos modelos resultantes da análise.

Os métodos estruturados conduzem à criação de três modelos distintos, cada um dos quais permite visualizar um aspecto particular do sistema: os dados, os processos e os eventos. Durante a fase de concepção, é necessário integrar e reestruturar estas três perspectivas, de modo a produzir um modelo para o sistema em desenvolvimento. Esta problemática parece não existir no desenvolvimento orientado ao objecto, pois os modelos resultantes da análise e da concepção, assim como as linguagens de programação orientada ao objecto, utilizam o mesmo conceito básico de modelação, o objecto. Cada objecto inclui os três aspectos dos sistemas, supramencionados, o que permite o mesmo poder de modelação, mas com a vantagem de possibilitar um maior nível de integração e relacionamento.

Nas metodologias estruturadas, a transição entre as fases de análise e de concepção não é tão suave, mas antes mais abrupta e explícita, o que permite distinguir com mais naturalidade a fronteira entre as duas fases. Esta transição bem acentuada resulta do facto de os modelos da análise não serem refinados para gerar os modelos da concepção. Em vez disso, na fase de concepção, são criados modelos novos, sem ligação formal e directa aos modelos criados na fase de análise, pelo que, nem sempre é possível garantir o relacionamento entre esses modelos. Este problema, conhecido como o *problema da continuidade dos modelos* [Kumar et al., 1996b,

[8] *Silver bullet* é o termo inglês usado na engenharia de software para referir uma eventual técnica, abordagem ou ferramenta que pudesse tornar simples o desenvolvimento de software.

pág. 6], torna-se ainda mais evidente, quando anteriormente se procedeu à validação dos modelos iniciais, já que os resultados obtidos para estes não podem ser aplicados, com garantia, aos modelos subsequentes, por não haver o tal relacionamento entre os modelos. O facto de a análise, a concepção e até a programação orientadas ao objecto usarem como base o mesmo modelo de objectos, o qual vai sendo refinado e melhorado, ao longo do desenvolvimento do sistema, permite assegurar que existe, de facto, uma continuidade dos modelos.

O uso dum meta-modelo orientado ao objecto permite que se explorem todas as facilidades das linguagens baseadas em objectos e orientadas ao objecto. Esse tipo de meta-modelo encoraja a reutilização, não só de objectos, mas também de sistemas inteiros, e possibilita que se desenvolvam gradualmente sistemas relativamente complexos, porque a integração está disseminada ao longo de todo o desenvolvimento. Embora não seja consensual, tem sido sugerido que os conceitos associados à tecnologia orientada ao objecto assemelham-se mais à forma de cognição humana:

> "*Many people who have no idea how a computer works find the idea of object-oriented systems quite natural.*" [Robson, 1981].

A arquitectura para computadores, idealizada por von Neumann, divide um computador em 5 partes principais: memória, unidade de controlo, unidade aritmética, entrada e saída. Esta arquitectura, ainda muito popular nos dias de hoje, teve uma influência decisiva, até aos anos 80, na forma como as linguagens, em primeiro lugar, e os métodos e as metodologias, posteriormente, foram concebidos. De facto, a divisão muito nítida entre dados e programas, que a arquitectura de von Neumann impõe, manteve-se nas linguagens, métodos e metodologias, o que obriga desnecessariamente os humanos a terem de moldar o seu pensamento ao modo como os computadores estão estruturados para executar as aplicações. Só com o advento da tecnologia orientada ao objecto é que esta tradição se rompeu finalmente. A seguinte afirmação vivifica esta perspectiva:

> "*(. . .) programming paradigms should not be developed in the same way as computer hardware paradigms, but rather as people think.*" [Jacobson et al., 1992, pág. 117].

A finalizar, refira-se que o autor não é partidário de posições intransigentes ou fundamentalistas relativamente aos méritos das duas abordagens aqui em comparação. Nesse sentido, as inúmeras técnicas que foram desenvolvidas para as metodologias estruturadas não devem ser liminarmente ignoradas, só pelo simples facto de se mudar de abordagem. Todas aquelas que demonstrem ser válidas e cuja aplicação possa ser enquadrada no desenvolvimento de sistemas orientados ao objecto, devem ser aproveitadas, sem qualquer tipo de condicionante. Por exemplo, os diagramas de contexto e as máquinas de estado são duas técnicas usadas nas metodologias estruturadas que são igualmente de indiscutível utilidade nas metodologias orientadas ao objecto [Douglass et al., 1998].

2.3.4 Vantagens e desvantagens do paradigma dos objectos

Vantagens

Apresenta-se, seguidamente, uma série de benefícios que advêm da utilização correcta do paradigma dos objectos, no desenvolvimento dum dado sistema [Graham, 1991, pág. 31–2]:

1. Objectos bem concebidos, em sistemas orientados ao objecto, formam a base para que outros sistemas possam ser desenvolvidos, a partir de módulos reutilizáveis, o que conduz a uma maior produtividade.
2. A comunicação entre objectos por mensagens significa que a interface entre módulos e os sistemas exteriores é mais simples.
3. A análise e a concepção baseadas na partição do sistema em objectos é, normalmente, considerada mais natural do que a decomposição funcional hierárquica.
4. O encapsulamento facilita a construção de sistemas mais seguros[9].
5. A programação orientada ao objecto e, em particular, o mecanismo de herança permitem que se definam e usem módulos incompletos, do ponto de vista funcional, podendo posteriormente estes ser completados, sem que isso implique necessariamente que haja interferência na operação de outros módulos ou dos seus clientes.
6. O desenvolvimento orientado ao objecto pode ser conjugado com o uso de métodos formais de especificação.
7. A orientação ao objecto é uma ferramenta para lidar com a complexidade dos sistemas.
8. Os sistemas orientados ao objecto escalam mais facilmente, ou seja, é possível mais rapidamente passar dum sistema pequeno para um idêntico de maiores dimensões.
9. Os problemas que surgem com a evolução do sistema, bem como com a sua manutenção, são atenuados, devido à forte partição que resulta do encapsulamento e da uniformização das interfaces.
10. Potencialmente, os sistemas orientados ao objecto conseguem mais naturalmente captar o significado (a semântica) duma aplicação, i.e. facilitam significativamente a compreensão do sistema.

Inicialmente, as metodologias orientadas ao objecto foram definidas para resolver problemas no domínio do software, pelo que, à primeira vista, pode parecer forçada a sua utilização em projectos de hardware. Contudo, a prática tem mostrado que o paradigma orientado ao objecto é aceite com relativa naturalidade pelos engenheiros de hardware, já que estes se habituaram, ao longo do seu percurso profissional, a manipular objectos (placas, componentes, módulos, células) que comunicam por sinais [Williams, 1992] [Fernandes e Machado, 1997]. A seguinte afirmação confirma esta perspectiva:

> "*The object-oriented view is a good way of thinking about hardware. Each device is an object that operates concurrently with other objects (other devices or software).*" [Rumbaugh et al., 1991, pág. 203].

Alguns autores vislumbram mesmo uma semelhança entre os objectos, tradicionalmente usados no software, e as máquinas de estados finitos, usualmente utilizadas no hardware [Mills, 1988] [Robinson, 1992] [Douglass, 1998]. Segundo essa perspectiva, um objecto pode ser visto como uma máquina de estados finitos que comunica com outras, em que o respectivo estado é alterado através da execução dos seus serviços.

Para reforçar esta ideia, nos últimos tempos, a *fertilização cruzada* (*cross-fertilization*) (i.e. a transferência de técnicas entre os domínios do software e do hardware) tem-se tornado cada vez mais comum, diluindo, um pouco mais, as, já de si difusas, fronteiras entre os dois domínios [Kumar et al., 1994]. Paralelamente, o *co-projecto de hardware/software* emergiu como uma das disciplinas que mais interesse tem despertado em diversas equipas de investigação

[9] Um sistema diz-se *seguro* (*safe*) se não cria acidentes que provoquem danos em pessoas ou bens.

[Kumar et al., 1996b] [Dias et al., 1996] [Yen e Wolf, 1996] [Bergé et al., 1997] [Gupta, 1997] [Esteves et al., 1997].

As metodologias orientadas ao objecto para desenvolvimento de software conheceram grande aceitação por permitirem gerir, com relativa facilidade, a complexidade dos sistemas e por aumentarem significativamente a reutilização. Estes dois conceitos, assim como a possibilidade de descrição a um nível mais elevado de abstracção, uma maior legibilidade e uma manutenção facilitada dos projectos, são os principais responsáveis pelo aparecimento de diversas propostas com o intuito de introduzir princípios orientados ao objecto no desenvolvimento de sistemas hardware [Müller e Rammig, 1989] [Verschueren, 1992] [Kumar et al., 1994] [Agsteiner et al., 1995] [Bergé et al., 1996] [Schumacher et al., 1996] [Morris et al., 1996].

A crescente vulgarização das linguagens de descrição de hardware (HDLs), nomeadamente VHDL (HDL oficial do ANSI e do IEEE) [IEEE, 1994], como formato privilegiado de especificação, tornou as funções dum engenheiro de hardware bastante semelhantes às dum engenheiro de software: ambos têm de descrever o sistema duma forma textual, seguindo a sintaxe da linguagem de implementação escolhida. Foi, portanto, como consequência natural que foram surgindo inúmeras extensões, em diversas direcções, para adicionar conceitos orientados ao objecto à linguagem VHDL [Glunz, 1991] [Perry, 1992] [Dunlop, 1994] [Schumacher e Nebel, 1995b] [Swamy et al., 1995] [Cabanis et al., 1996] [Benzakki e Djafri, 1997]. Nenhuma destas extensões foi reconhecida como oficial, embora seja de esperar que, mais tarde ou mais cedo, o IEEE inclua princípios do paradigma dos objectos na linguagem VHDL.

A criação do *IEEE DASC Object-Oriented VHDL Study Group*, em finais do ano de 1993, veio dar resposta ao crescente interesse na aplicação de técnicas orientadas ao objecto, durante a elaboração de especificações escritas em VHDL ("modelos VHDL" na terminologia própria da linguagem). Os objectivos principais desse grupo de estudo consistem, essencialmente, nos seguintes aspectos relacionados com "VHDL orientado ao objecto": (1) clarificar o que pode significar; (2) definir a sintaxe e respectiva semântica; (3) indicar como pode ser usado; e (4) enumerar as vantagens decorrentes da sua utilização. Alguns dos requisitos necessários para estender a linguagem VHDL com conceitos da orientação ao objecto já foram identificados, com base em entrevistas realizadas, na Europa, em companhias de telecomunicações e em *software houses* [Putzke-Raming et al., 1997]. Existem outras linguagens que também permitem que se possam modelar sistemas hardware, utilizando técnicas orientadas ao objecto [Schumacher e Nebel, 1995a].

É conhecido que os progressos verificados na engenharia de software se fizeram, quase sempre, de baixo para cima (das linguagens para as metodologias). A engenharia de software dedicou-se, nos seus primórdios, à definição de técnicas na área da programação, com especial ênfase nas linguagens de programação e na compilação. A evolução trouxe, posteriormente, novos tópicos de interesse, primeiramente os métodos de concepção, mais tarde, os métodos de análise e, finalmente, as metodologias de desenvolvimento. Curiosamente, a evolução da "orientação ao objecto" conheceu um progresso semelhante ao atrás descrito. Igualmente, após a introdução das HDLs, o percurso trilhado na engenharia de hardware foi muito semelhante: primeiro apareceram as linguagens de descrição, mais tarde as extensões orientadas ao objecto e agora começa a surgir a percepção de que a investigação dos próximos tempos se centrará na definição de métodos e metodologias (orientados ao objecto) para desenvolvimento de sistemas hardware.

Nesta linha de pensamento, Ecker e Mrva prevêem que a *orientação ao objecto* constituirá o próximo salto qualitativo ("*quantum leap*") que o projecto de hardware vai conhecer, como a transição para elementos digitais ou a introdução das HDLs também o haviam sido anterior-

mente [Ecker e Mrva, 1996]. A razão para esta previsão reside, segundo aqueles autores, não só nos resultados desanimadores ou pouco promissores que se conseguiram obter com outras abordagens, mas essencialmente no contributo decisivo que o paradigma do objecto teve no aumento da produtividade no desenvolvimento de software. Adicionalmente, os conceitos associados a este paradigma têm as suas origens na modelação e na simulação de sistemas, o que se revela uma vantagem óbvia.

As eventuais vantagens que resultam da aplicação de técnicas orientadas ao objecto no domínio do hardware são as que a seguir se enumeram [Kumar et al., 1996a]:

1. Modificação e manutenção facilitadas das especificações.
2. Instanciação fácil de componentes com parâmetros diferentes.
3. Adaptação de componentes genéricos para criar componentes mais especializados.
4. Composição rápida de novos componentes.
5. Identificação e reutilização de componentes comuns.
6. Possibilidade de utilizar técnicas de síntese e verificação já experimentadas em sistemas de software.

A utilização do objecto como elemento de modelação, ao longo de todo o desenvolvimento, permite um mais fácil relacionamento entre a solução final e os requisitos [Smith e Tockey, 1988]. Adicionalmente, as metodologias orientadas ao objecto, tanto na fase de análise como na fase de concepção, manipulam modelos com um grau de estabilidade muito superior ao das metodologias estruturadas, relativamente a alterações surgidas na especificação do problema [Coad e Yourdon, 1991].

Os requisitos do sistema mudam frequentemente ao longo do desenvolvimento, muitas vezes, porque a simples existência dum projecto altera as regras do problema [Booch, 1991, pág. 4]. O contacto com protótipos ou mesmo com o produto final faz com que os utilizadores repensem as suas necessidades e expectativas em relação ao sistema. Por outro lado, à medida que o projecto vai avançando, a equipa de projecto vai ganhando um conhecimento mais profundo sobre o domínio de aplicação, o que lhe possibilita captar mais facilmente alguns aspectos do sistema.

Assim, um projectista dum sistema, durante o ciclo de vida deste, deve estar preparado para lidar com várias alterações, adaptando-se à evolução das necessidades dos seus utilizadores finais. Vários factores podem condicionar esta evolução: a exigência dos utilizadores, a concorrência entre empresas, a tecnologia, a legislação, etc. Por este motivo, devem utilizar-se mecanismos que permitam modelar os aspectos do sistema mais resistentes à mudança, que assim poderão ser usados como suporte às restantes fases de desenvolvimento. As alterações nos requisitos dum problema devem ser consideradas como um facto natural e não como o resultado dum pedido inicial mal elaborado, até porque, como diz o adágio, "o cliente tem sempre razão". Esta ideia pode ser resumida na seguinte afirmação:

> "*We have to accept changing requirements as a fact of life, and not condemn them as a product of sloppy thinking.*" [Fischer, 1989].

Assim, a postura habitual de muitas equipas de projecto em tentar "congelar" os requisitos, para a partir deles poder avançar nas fases seguintes do desenvolvimento é errada e inaceitável, embora desculpável por facilitar a sua tarefa. Em alternativa, o que é necessário é encontrar os mecanismos adequados que permitam incorporar alterações nos requisitos, sem grandes perturbações no trabalho anteriormente desenvolvido. Como se sabe, os sistemas actuais estão

sujeitos a alterações constantes, sendo o seu aspecto mais volátil precisamente a sua funcionalidade. Assim, tem havido uma maior preocupação em reduzir a volatilidade dos modelos obtidos durante a análise, vincando os aspectos dos sistemas que apresentam uma maior resistência à mudança, no sentido de diminuir a necessidade de alteração desses mesmos modelos durante a análise e, principalmente, do próprio sistema quando já em pleno funcionamento.

Uma das grandes vantagens duma metodologia orientada ao objecto para desenvolvimento de sistemas reside no conjunto de mecanismos que é disponibilizado para modelar o mundo real, o que facilita a manutenção e permite obter um conhecimento mais profundo dos sistemas, cuja complexidade seja superior àquela que a capacidade mental dum ser humano "normal" consegue captar.

Como indicado na subsecção 2.5.2, o conceito de representação unificada apresenta várias vantagens no desenvolvimento de sistemas hardware/software. Pela descrição já feita, os metamodelos usados nas metodologias orientadas ao objecto incluem algumas das características que os permitem tornar-se numa representação unificada para o desenvolvimento de sistemas embebidos complexos, uma vez que disponibilizam uma série de mecanismos de modelação que facilitam a gestão da complexidade. É exactamente essa possibilidade que se pretende demonstrar neste trabalho.

Desvantagens

Não obstante a abordagem orientada ao objecto apresentar um conjunto bem significativo de vantagens, cabe também realçar os problemas que advêm da sua utilização ou, pelo menos, as circunstâncias em que as vantagens não podem ser aproveitadas.

Em primeiro lugar, relativamente à notação há que referir, basicamente, duas questões: (1) o tamanho dos modelos, normalmente, muito grande, mesmo para sistemas de média complexidade; e (2) o elevado número de vistas e diagramas a criar e manter. Relacionado com este último problema, levanta-se ainda uma outra questão que se refere à forma como se garante que as várias vistas criadas estão coerentes entre si.

Por outro lado, as metodologias orientadas ao objecto dão, dum modo generalizado, cobertura total à fase de análise, mas, quase sempre, são menos completas nas fases de concepção[10] e, principalmente, de implementação. Este último problema não é, contudo, exclusivo das metodologias orientadas ao objecto, verificando-se igualmente nas metodologias estruturadas.

A reutilização nem sempre é explorada, pois a concepção de módulos reutilizáveis, a colocar em bibliotecas para uso futuro, faz aumentar o custo e o tempo de desenvolvimento do sistema actual, embora permita que os sistemas seguintes sejam menos dispendiosos. Construir bibliotecas de componentes é, pois, um investimento a médio prazo, não trazendo benefícios evidentes de forma imediata. Assim, há normalmente a tentação em realizar o sistema, da forma mais rápida possível, à custa da oportunidade de reutilização futura. Este problema é ainda acentuado pela inexistência de bibliotecas de componentes, comercialmente disponíveis, o que obriga o projectista a desenvolver todos os módulos de que precisa. Um outro problema, inibidor da reutilização, reside na necessidade em conhecer profundamente a biblioteca de classes para poder explorar, ao máximo, as capacidades da sua utilização. De facto, para usar eficientemente

[10]Por exemplo, a metodologia Booch'91, em comparação com a metodologia OMT, coloca menos ênfase na análise e aborda com mais profundidade a concepção. O próprio Booch viria a admitir esse facto ao afirmar: "*The Booch Method as it was defined in 1990 was admittedly weak with regard to analysis.*" [Booch, 1996, pág. 111].

componentes de uma biblioteca, é essencial estar-se familiarizado com a biblioteca, o que requer o investimento de muito tempo a avaliar as classes, bem como as respectivas operações e os respectivos atributos.

Relativamente às linguagens orientadas ao objecto, apesar da maioria delas suportar todas as características desejáveis para implementar sistemas segundo a abordagem orientada ao objecto, apresentam também algumas desvantagens. Por exemplo, o mecanismo de *ligação dinâmica*, necessário para determinar qual o procedimento (pedaço de código) a executar para um dado serviço, tem um tempo de execução mais elevado, se comparado com uma chamada duma função.

A aplicação directa das metodologias orientadas ao objectos para software aos sistemas hardware, em geral, e ao sistemas embebidos, em particular, apresenta, também, alguns problemas que a seguir se descrevem

Em primeiro lugar, a metodologia deve ter explicitamente identificado um passo (e respectivas tarefas e técnicas), cujo objectivo principal seja a *partição* do sistema em componentes hardware e software. A partição não é considerada uma tarefa importante em projectos de software, porque se parte do princípio que todo o sistema será compilado para executar numa dada plataforma pré-definida, ou seja, a componente hardware está perfeitamente identificada, não havendo necessidade em defini-la e projectá-la. Refira-se que, tradicionalmente, o co-projecto centrava a sua acção essencialmente na partição do sistema em componentes hardware e software, a fim de produzir uma implementação o mais óptima possível, partindo do princípio que a especificação do sistema estava correcta [Morris et al., 1996]. Esta visão, se bem que ainda perdure em algumas pessoas, tem, felizmente, vindo a diluir-se, o que significa que o co-projecto deve também contemplar como fundamentais as actividades de levantamento de requisitos, a análise e o teste.

A metodologia deve também identificar e, se necessário, desenvolver os sistemas que não fornecem directamente a funcionalidade do sistema, mas que incluem componentes que a fornecem. Neste apartado, está incluído, por exemplo, o sistema operativo.

2.4 A análise no desenvolvimento de sistemas

Depois de se ter dado uma perspectiva abrangente do desenvolvimento de sistemas, pretende focalizar-se, nesta secção, algumas questões que dizem respeito à análise de sistemas, uma vez que foi nesta fase do processo de desenvolvimento que se verificou o contributo mais significativo deste trabalho. A discussão pretende-se o mais genérica possível (i.e. independentemente dos sistemas serem implementados em software ou em hardware), o que só vem reafirmar o pressuposto que constitui uma das teses deste trabalho: *o hardware e o software são logicamente equivalentes*.

2.4.1 Considerações genéricas

Sucintamente, pode definir-se a *análise* como o estudo do problema, antes de tomar qualquer acção [DeMarco, 1979]. A análise debruça-se sobre o estudo do domínio da aplicação, conduzindo a uma especificação do comportamento exteriormente visível. Essa especificação deve descrever duma forma completa, consistente e legível o sistema que se pretende construir, cobrindo quer a funcionalidade requerida, quer as características operacionais (exemplo: fiabil-

idade, disponibilidade, desempenho). A especificação trata o sistema como uma "caixa negra", indicando todas as características do seu comportamento externo, mas ignorando os aspectos da estrutura interna que permitem a obtenção desse comportamento [Zave, 1984]. A análise determina o que o sistema deve fazer para satisfazer o cliente e não a forma como o sistema deve ser construído [Coad e Yourdon, 1991, pág. 19]. Assim, analisar um sistema consiste em modelá-lo sem ter em consideração a plataforma de implementação a usar.

A análise de sistemas é conduzida com os seguintes objectivos [Pressman, 1997, pág. 269]:

- Identificar as necessidades do cliente.
- Avaliar a viabilidade do sistema.
- Analisar o sistema do ponto de vista económico e técnico.
- Atribuir funcionalidades ao hardware, ao software, às pessoas e aos demais elementos do sistema.
- Estabelecer custos e restrições temporais.
- Criar uma especificação do sistema que sirva de base para as fases seguintes de projecto.

A perspectiva do autor, relativamente aos objectivos da análise, é menos abrangente que esta, uma vez que alguns dos objectivos referidos pressupõem actividades que podem ser posicionadas em fases anteriores ao desenvolvimento (estudos de viabilidade).

O estudo de viabilidade tem por objectivo antecipar cenários previsíveis no desenvolvimento do software, mas, em princípio, não deve realizar qualquer actividade de desenvolvimento. Como resultado da sua realização devem emergir os três seguintes pontos:

1. Definição do problema.
2. Soluções alternativas e os respectivos benefícios.
3. Recursos, custos e datas em relação às alternativas propostas.

A análise de sistemas inicia-se usualmente com o estudo do enunciado do problema, que explicita, entre outros, os *requisitos* pretendidos. O documento com o enunciado do problema varia (em forma, em tamanho e em conteúdo) de projecto para projecto. Não deve, portanto, o analista estar à espera dum dado formato para o documento, mas antes estar preparado para qualquer tipo de documento e, independentemente do conteúdo ou formato, basear-se naquele para cumprir a sua missão.

Tornou-se relativamente óbvio que a definição clara dos requisitos do sistema tem um impacto enorme na qualidade e utilidade finais deste, bem como na eficiência do processo de desenvolvimento associado. Aliás, como resultado duma investigação, que incluiu dados de 107 projectos realizados em 70 organizações britânicas, chegou-se à conclusão que quanto maior for o tempo dispendido no levantamento de requisitos, menor tempo é depois necessário para todo o processo de desenvolvimento[11] [Chatzoglou e Macaulay, 1995]. Assim, depois de estudado o enunciado do problema, deve, sempre que possível, promover-se a realização duma série de entrevistas e discussões com diversas pessoas ligadas, de alguma forma, ao sistema, nomeadamente clientes, utilizadores e fabricantes, pois dificilmente um projecto constitui um êxito completo, se todas as pessoas que nele estiverem envolvidas não derem a sua aprovação [Zave, 1982].

O resultado final da análise consiste num modelo do sistema, que permite, entre outros, proceder às seguintes actividades:

- Visualizar os aspectos relevantes do sistema.
- Proceder à sua manipulação de modo a ser possível prever o efeito de eventuais alterações.

[11] Como corolário, pode afirmar-se que quanto mais dinheiro se investir durante o levantamento de requisitos, mais barato é o desenvolvimento do sistema.

- Comunicar, através das ideias nele expressas, com os especialistas do domínio de aplicação.
- Reduzir a complexidade[12].

Durante a fase de análise, o maior esforço é direccionado na aquisição de conhecimento relativamente aos aspectos mais importantes do sistema. Os analistas, em conjunto com os utilizadores, funcionários ou gestores, devem identificar o *que* o sistema faz, sem se preocuparem em *como* é que ele é construído. Para tal, os analistas recorrem a questionários, entrevistas, mesas redondas, à observação das actividades desenvolvidas e análise de manuais, relatórios, formulários e outros documentos para recolherem as informações de que precisam[13]. É normalmente difícil, embora totalmente desejável, manter os requisitos estritamente separados das suas próprias soluções. Infelizmente, na maioria dos casos, os documentos produzidos em fases iniciais do desenvolvimento misturam os requisitos e as soluções.

No decurso da sua actividade, os analistas deparam-se com diversos problemas. Alguns dos problemas mais frequentemente enfrentados pelos analistas são os seguintes [Barker, 1990] [Sully, 1993]:

1. A dificuldade em comunicar com os especialistas da área.
2. A dificuldade em compreender, com rigor e exactidão, o sistema a desenvolver.
3. As contínuas modificações a que pode estar sujeito o sistema em estudo.
4. A duplicação de esforço resultante de não se reaproveitarem resultados de análises anteriores.

É comum existirem problemas de comunicação entre os analistas do sistema e os especialistas do domínio de aplicação [Booch, 1991, pág. 4]. A estes é, geralmente, muito custoso dar àqueles uma expressão precisa, numa forma perceptível, das suas reais necessidades relativamente ao sistema a desenvolver. Além disso, em muitas situações, os especialistas têm apenas uma vaga ideia do que pretendem que o sistema faça. Estes problemas não são da responsabilidade de nenhuma das duas partes, pois eles existem por cada uma das partes não ter o mesmo tipo de sensibilidade e conhecimento relativamente ao problema em causa. Os especialistas e os analistas têm perspectivas diferentes em relação à natureza do problema, pelo que também fazem leituras diferentes. Mesmo que os especialistas tenham um conhecimento total das suas necessidades, o que nem sempre sucede, não é simples passar essa informação para os analistas, consistindo a forma mais comum de veicular essa informação através duma série de documentos de texto, por ventura, acompanhados de alguns gráficos. Estes documentos, que vulgarmente se apelidam de "papelada", são difíceis de entender e susceptíveis de ser interpretados de forma pessoal.

Outro dos problemas enfrentados reside em identificar os aspectos que são realmente relevantes para o sistema em causa. Para que tal se concretize, o analista deve possuir métodos e técnicas que orientem o seu estudo e lhe possibilitem lidar com a complexidade crescente dos sistemas que actualmente há interesse em desenvolver, de modo a obter um conhecimento profundo dos mesmos. Neste sentido, os métodos de análise, seguindo uma decomposição funcional ou orientada ao objecto, são duas das ferramentas a que o analista pode recorrer nos dias de hoje[14].

[12] Esta é a principal razão para a construção de modelos, que, por focarem apenas nos aspectos relevantes do sistema, abrangem uma quantidade limitada de informação, permitindo assim reduzir a complexidade dos sistemas em análise.

[13] Além das citadas, existem várias técnicas e abordagens para adquirir os requisitos do sistema, mas o seu estudo está fora do âmbito deste trabalho. Alguns livros tratam este assunto em profundidade, como por exemplo, [Davis, 1994] [Loucopoulos e Karakostas, 1995] [Macaulay, 1996].

[14] Existem outras abordagens para analisar um problema, mas, por uma questão de simplicidade, consideraram-se apenas estas duas.

A qualidade do trabalho realizado pelo analista tem um impacto decisivo nas restantes fases de desenvolvimento. É crucial que não contenha erros, pois estes poderão ser os responsáveis por erros produzidos nas restantes fases, o que implica correcções de alto custo. Um modelo correcto dum sistema, onde se encontrem representados todos os aspectos relevantes, permite reduzir erros futuros, além de facilitar o trabalho das pessoas que vão conceber e implementar o sistema. Mais de metade dos erros cometidos são usualmente introduzidos durante o levantamento de requisitos [Kit, 1995, pág. 19]. Além disso, os custos associados à remoção dos erros são minimizados se forem detectados na mesma fase em que foram introduzidos. Neste contexto, o levantamento de requisitos deve ser considerado como uma tarefa crucial nos projectos de sistemas informáticos, sendo importante investir na melhoria das técnicas que lhe estão associadas [Martins, 1998].

Para captar os requisitos do utilizador, o projectista deve conhecer um conjunto, o mais alargado possível, de técnicas (*portfolio*) e seleccionar e aplicar aquelas que melhor se adaptam a cada situação [Macaulay, 1996, pág. 127]. Esta autora considera a existência de 7 situações típicas de desenvolvimento, caracterizadas pela relação cliente-fornecedor e pelo processo associado. A ideia é que as técnicas necessárias, para cada uma das situações, dependem fortemente dessas duas componentes.

A análise orientada ao objecto estuda os requisitos do sistema, na perspectiva das classes e dos objectos encontrados no vocabulário específico do domínio de aplicação. Os requisitos do utilizador devem ser expressos na terminologia do problema, definindo claramente o que os utilizadores querem que o sistema faça. Por outras palavras, os requisitos do utilizador devem ser definidos segundo uma perspectiva operacional [Stevens et al., 1998, pág. 20]. Nos últimos anos, têm surgido diversos métodos de análise orientada ao objecto, o que demonstra claramente o entusiasmo surgido em redor desta abordagem. Os seus defensores referem frequentemente algumas vantagens a ela associadas:

- Representa uma forma mais natural de estudo dos sistemas.
- Permite uma transição mais suave para a fase de concepção.
- Fomenta a reutilização de modelos criados para outros sistemas.
- Possibilita o estabelecimento duma correspondência directa entre os objectos reais e os componentes do modelo resultante da análise.
- Encoraja o analista a concentrar-se no que o sistema deve fazer, em detrimento de como tal é conseguido.

Por exemplo, o método OOA, que se dedica, em exclusivo, à análise de sistemas baseando-se numa abordagem orientada ao objecto, apresenta-se dividido em 5 etapas [Coad e Yourdon, 1991, pág. 34]:

1. Encontrar classes e objectos.
2. Identificar estruturas.
3. Identificar assuntos.
4. Definir atributos.
5. Definir operações[15].

Estas etapas devem ser entendidas como actividades, o que significa que a sua ordem não é fixa, embora aquela que é apresentada constitua a aplicação mais frequente do método.

Identificar os objectos numa determinada aplicação é considerada uma tarefa bem complicada, apesar de poder parecer relativamente simples, à primeira impressão. É uma questão que pro-

[15]Serviços, na terminologia do método.

duz, na maior parte dos casos, imensa discussão filosófica e introduz bastantes divergências nas equipas de projecto [Graham, 1991, pág. 249–63]. Os objectos podem ser identificados de várias formas, por um analista familiarizado com a aplicação, e escolher uma representação adequada é uma das tarefas de desenvolvimento que dificilmente pode ser realizada, sem intervenção humana. Uma estratégia habitualmente usada, para identificar os objectos relevantes dum determinado sistema, assim como os atributos e as operações daqueles, consiste em extrair os substantivos e os verbos da especificação em língua natural, em que os objectos e os atributos correspondem aos substantivos e as operações aos verbos [Abbott, 1983]. Esta estratégia tem de ser usada com algum critério, pois é sempre possível transformar um substantivo num verbo e *vice-versa* (exemplo: "a leitura é interrompida às 15h00" pode ser transformada em "a interrupção da leitura verifica-se às 15h00").

Estudos recentes mostram que o número de desenvolvimentos de raiz de sistemas é, actualmente, menor que o número de modernizações e alterações que se fazem a sistemas já em funcionamento. A comprová-lo estão as adaptações que inúmeras organizações tiveram que fazer aos seus sistemas informáticos, devido ao *bug* do ano 2000 e à transição para a nova moeda europeia (Euro). Assim, é frequente as organizações sentirem a necessidade em modernizar, melhorar, adaptar ou reformular um sistema já em funcionamento. Se bem que é perfeitamente admissível enquadrar-se esta actividade na fase de manutenção, é mais apropriado considerar que o sistema será novamente desenvolvido, caso sejam necessárias novas técnicas (por exemplo, recorrendo a uma abordagem orientada ao objecto em substituição duma abordagem estruturada). Nesta situação, a documentação existente para o sistema actual pode (e deve) funcionar como base para a análise a realizar para o novo desenvolvimento. Esta técnica, conhecida por *engenharia reversa* ou *reengenharia*, será abordada neste trabalho, no âmbito da aplicação da metodologia MIDAS a um caso prático (secção 7.2).

2.4.2 Requisitos e especificações

Muitos dos sistemas a desenvolver têm os respectivos requisitos escritos em língua natural, pelo que é aceitável considerar-se que, regra geral, os documentos que descrevem esses requisitos são ambíguos, incompletos e incoerentes. Este facto motiva a necessidade de transformar os requisitos, usando um dado *meta-modelo*, de forma a especificar ou modelar com maior precisão e clareza a funcionalidade pretendida para o sistema em causa.

Cada meta-modelo disponibiliza um conjunto de peças e regras para composição dessas peças, de modo a facilitar a criação duma descrição da funcionalidade do sistema. Um meta-modelo apresenta-se útil se possuir determinadas propriedades [Gajski et al., 1994, pág. 16]. Um meta-modelo deve ser *formal*, no sentido de ser claro, não apresentando ambiguidades; deve ainda ser *completo*, permitindo descrever inteiramente o sistema; e deve também ser *legível* para os projectistas (e eventualmente para os clientes) que o usam, facilitando a compreensão da funcionalidade do sistema.

O projectista escolhe um dado meta-modelo para decompor o sistema em vários sub-sistemas (peças) e, depois, cria uma *especificação* que descreve essas peças e respectivas interligações, usando, para esse fim, uma dada linguagem ou notação[16]. Uma determinada linguagem tem associado um meta-modelo, ao passo que um meta-modelo pode ser captado por várias lingua-

[16]Muitas vezes, a escolha do meta-modelo não é, infelizmente, a mais adequada, por não ser feita de forma ponderada ou consciente; umas vezes por desconhecimento de outras alternativas, outras por comodismo ou ainda por falta de disponibilidade de ferramentas apropriadas.

gens.

Geralmente, os modelos são uma aproximação do sistema representado e, como consequência, um modelo nem sempre descreve todas as propriedades do sistema. Na maioria dos casos, alguns pressupostos são feitos na construção dos modelos e a validade daqueles influencia directamente a forma como o modelo caracteriza o sistema. Contudo, é importante classificar um modelo, não necessariamente em termos de correcto ou incorrecto, mas antes em termos da sua adequação para revelar certos aspectos acerca do sistema [Kumar et al., 1996b, pág. 116].

Uma vez que não há, actualmente, disponível nenhuma linguagem formal para exprimir completamente os requisitos dum sistema, a transformação dos requisitos pretendidos numa especificação é considerada uma tarefa bem complexa, até mesmo, uma arte. Para ser útil e eficiente, uma dada especificação tem de satisfazer um conjunto de propriedades [Calvez, 1996].

Nas disciplinas mais antigas da engenharia, a palavra "especificação" tem um significado bem preciso. Na engenharia de sistemas computacionais, porém, o termo é usado, em contextos distintos, com significados distintos. Em termos genéricos, pode ver-se uma especificação como um acordo entre o produtor dum serviço e o respectivo consumidor (ou entre o fabricante e o utilizador). Dependendo do contexto, pode, para cada fabricante e utilizador diferentes, obter-se uma especificação de natureza diferente, o que significa que, durante as diversas fases do processo de desenvolvimento, são usadas várias especificações [Ghezzi et al., 1991, pág. 151]:

- Uma *especificação de requisitos* é um acordo entre o utilizador final e o engenheiro responsável pelo desenvolvimento do sistema.
- Uma *especificação de concepção* é um acordo entre o arquitecto do sistema e os engenheiros responsáveis pela implementação do sistema.
- Uma *especificação de módulo* é um acordo entre os programadores que usam um módulo e o programador que o implementou.

Uma *especificação* é definida como uma descrição precisa da funcionalidade do sistema, que satisfaz os respectivos requisitos. Uma especificação dum sistema deve indicar o que o sistema faz e não como tal é conseguido. Idealmente, uma especificação deve ser completa, coerente, legível, relacionável com os requisitos, formal, executável, clara, compreensível e modificável. Se alguns destes adjectivos não são susceptíveis de criar dúvidas de interpretação, outros há que carecem duma definição precisa, a fim de esclarecer o seu significado no presente contexto.

Uma *especificação executável* é aquela que pode ser manipulada por ferramentas automáticas, a fim de ser processado algum tipo de informação útil ao seu desenvolvimento. Uma especificação executável é mais do que um mero modelo estático em papel que define o comportamento dum sistema, devendo permitir, relativamente a este, a simulação ou animação da sua funcionalidade, a verificação de algumas das suas propriedades, a exploração de alternativas de modelação ou a sua síntese automática (compilação). Alguns dos benefícios observados na utilização de especificações executáveis, em diversos projectos de grande dimensão, podem ser encontrados em [Harel, 1992].

Para que seja possível executar uma especificação, esta deve estar expressa numa linguagem formal, uma vez que assim a sua semântica é bem definida, deixando portanto de haver ambiguidades na sua interpretação. Como definido anteriormente, uma linguagem é formal se tiver associado um meta-modelo formal (formalismo).

O uso de especificações executáveis tem associada uma *abordagem operacional*, que foi iniciada nos princípios dos anos 80 por diversas equipas de investigação, tendo conhecido a sua expressão mais clara no trabalho desenvolvido por Zave que também criou PAISLEY, uma linguagem e

um ambiente especialmente concebidos para a criação de especificações executáveis [Zave, 1984].

Na abordagem operacional, ao tratarem-se todos os modelos usados como programas escritos numa linguagem de modelação de alto nível, torna-se possível a execução daqueles. O termo "executável" significa, neste contexto, quando aplicado a uma especificação, que esta pode ser compilada, de que resulta um programa que pode ser executado. Esta possibilidade de se executarem as especificações facilita a eliminação dum dos grandes problemas da abordagem tradicional, que resulta da ânsia dos projectistas em sentir se o sistema que está a ser desenvolvido coincide ou não com aquele que o cliente realmente pretende.

Se as especificações não forem executáveis, podem discutir-se e analisar-se, mas não pode ser simulado o comportamento que descrevem, pelo que as equipas de projecto são, naturalmente, impelidas para a fase de codificação, fazendo com que as fases anteriores sejam apressadas, o que raramente é desejável. O uso da abordagem operacional permite que as especificações executáveis possam ser usadas como protótipos, segundo o espírito referido na pág. 29.

A fig. 2.11 mostra as várias alternativas, segundo as quais se podem estudar as propriedades dum determinado sistema [Law e Kelton, 1991, pág. 4].

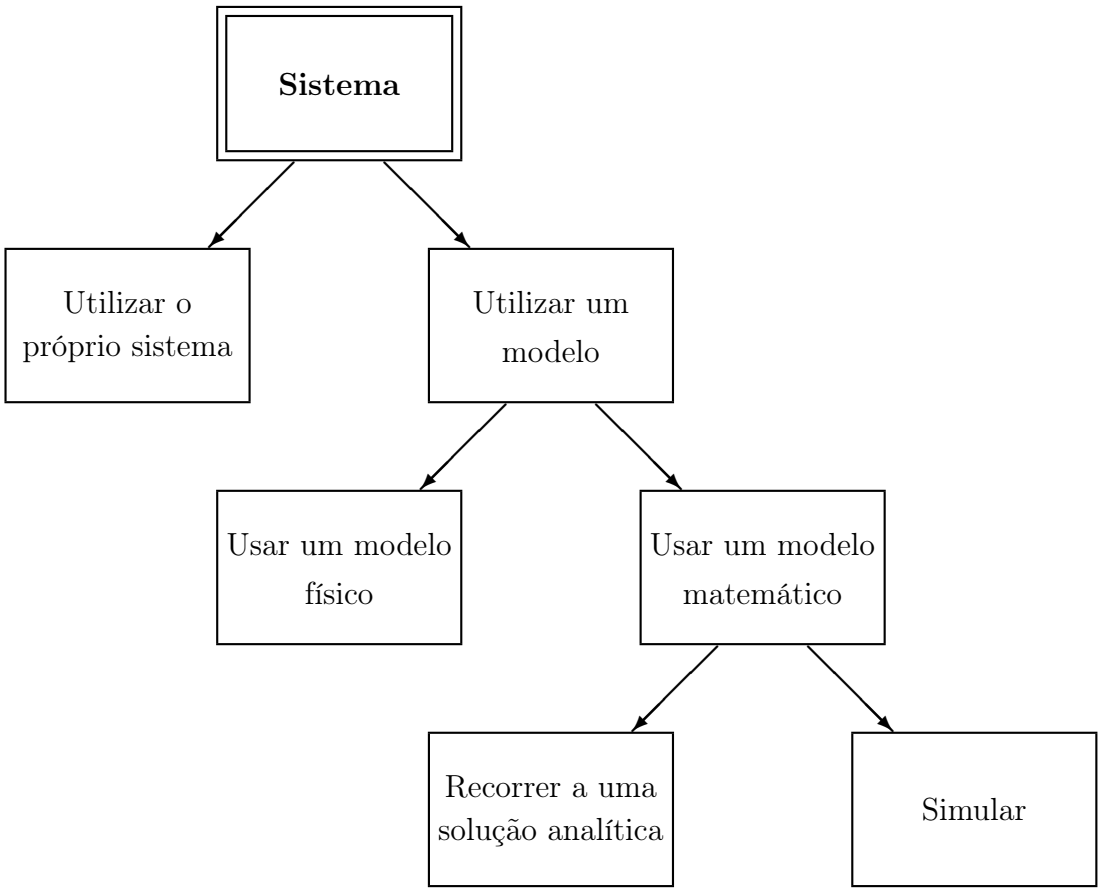


Figura 2.11: As várias alternativas para estudar as propriedades dum sistema.

- **Dilema entre experimentar o sistema actual vs. experimentar o modelo**
  Se for possível alterar fisicamente o sistema e fazê-lo operar sob as novas condições, essa será possivelmente a solução mais adequada. Contudo, raramente tal é possível, por ser muito custoso ou mesmo destrutivo. Por esse motivo, é usualmente preferível construir um *modelo* como uma representação do sistema e estudá-lo como um substituto do sistema. O uso dum modelo levanta sempre a questão da *adequabilidade do modelo* relativamente ao estudo pretendido para o sistema.
- **Dilema entre modelo físico vs. modelo matemático**
  Quando se constrói um túnel de vento para testes aerodinâmicos, uma miniatura dum edifício ou ainda o painel dum avião para treino de voo, está-se perante *modelos físicos*. Este tipo de modelo tem uma utilização pouco habitual, ao contrário do que sucede com os *modelos matemáticos*. Estes representam o sistema em termos de relações lógicas e quantitativas, que são manipuladas para verificar como o modelo reage e, por consequência, como o sistema reagiria, caso este seja bem representado por aquele. Um exemplo típico dum modelo matemático é a lei de Ohm ($V = R.I$), que relaciona a tensão eléctrica com a resistência e a intensidade de corrente.
- **Dilema entre solução analítica vs. simulação**
  Uma vez construído o modelo matemático, há que examiná-lo para verificar como ele pode ser usado para responder a questões interessantes sobre o sistema que representa. Se o modelo é relativamente simples, então pode trabalhar-se o conjunto de relações para obter uma *solução analítica* (exacta). Para a expressão dada na lei de Ohm, pode obter-se o valor da resistência, sabidos os valores da tensão e da intensidade. Porém, para sistemas complexos, os respectivos modelos matemáticos serão também complexos, eliminando assim qualquer possibilidade duma solução analítica. Neste caso, o modelo deve ser analisado por *simulação*, i.e. exercitando numericamente as entradas do modelo para determinar como são afectadas as suas saídas.

A *simulação* é uma das várias técnicas possíveis que os projectistas podem usar para, durante a vida útil dum sistema, o estudar, obtendo assim alguns dados relativos às suas propriedades (o desempenho, por exemplo). A adopção da abordagem operacional e, consequentemente, de especificações executáveis pressupõe o estudo dos sistemas usando a simulação, o que, apesar das limitações que lhe estão subjacentes, é, para a maioria dos sistemas complexos, uma das poucas alternativas viáveis.

Por outro lado, o uso duma especificação executável exige um esforço maior nas fases iniciais, se comparado com a abordagem tradicional. Para descrever um modelo executável, é necessário completar alguns aspectos que possam faltar ou estar ambíguos nos requisitos. Ora, os propósitos da análise são precisamente esses, pelo que esse esforço é compensado, já que a detecção de falhas vai tendo um custo cada vez mais avultado à medida que o desenvolvimento avança [Selic et al., 1994, pág. 41].

Para sistemas complexos, as dificuldades em obter uma especificação completa fazem com que a utilização duma abordagem incremental se revele extremamente importante, uma vez que permite iniciar o processo com uma especificação incompleta, que vai sendo adaptada e melhorada, ao longo de várias iterações (cf. modelo de processo incremental apresentado na subsecção 2.2.3).

A descrição dos requisitos, escrita em língua natural, está usualmente organizada como uma lista de cenários (do tipo "se acontecer isto, então fazer aquilo"), ao passo que a respectiva especificação é descrita como um algoritmo que tem forçosamente de abarcar todas as hipóteses

possíveis. Por outro lado, as descrições dos requisitos realçam o comportamento principal como um conjunto de cenários, omitindo alguns pormenores, o que obriga o leitor a inferir o restante comportamento[17]. Esta situação não se verifica com especificações, pois estas descrevem pormenorizadamente o comportamento, sem deixar grande espaço para interpretações de carácter subjectivo ou pessoal, pois as respectivas notações têm associada uma semântica precisa.

Ao criar-se uma *especificação formal*, o projectista é "obrigado" a considerar o funcionamento completo do sistema, o que, muitas vezes, o leva a descobrir alguns aspectos da funcionalidade do sistema que lhe haviam passado desapercebidos ou nos quais nem sequer havia pensado anteriormente. Adicionalmente, após completar a especificação formal (e mesmo antes, em alguns casos), o projectista pode colocar uma série de questões sobre o funcionamento do sistema e obter, através duma apreciação da especificação, as respectivas respostas. A apreciação referida pode ser feita por inspecção visual, por simulação, ou por validação automática das suas propriedades[18]. Uma especificação dos requisitos só permite responder directamente às questões relacionadas com os cenários considerados, uma vez que não é geralmente possível extrapolar comportamentos relativamente a situações não previstas. Finalmente, uma especificação, sendo executável, permite, como atrás se referiu, a simulação do sistema, para verificar a correcção da respectiva funcionalidade. Numa abordagem tradicional (em papel), tal só seria possível em fases adiantadas do processo (usualmente, ao nível da solução final).

Normalmente, são criadas, no mínimo, duas especificações durante o desenvolvimento dum sistema. Uma primeira, realizada durante a fase de análise, consiste num modelo do sistema totalmente independente da implementação e cuja função principal é permitir a comunicação com os especialistas no domínio da aplicação. Uma segunda, desenvolvida na fase de implementação, em que são consideradas as particularidades da plataforma alvo considerada.

A realidade é, todavia, bem diferente e, devido aos apertados prazos de entrega, a equipa de projecto começa, usualmente, o desenvolvimento do sistema directamente na fase de implementação. A análise é erradamente vista, nessas situações, como uma tarefa cujo resultado é unicamente atrasar a codificação do sistema [Wilson, 1987]. Esta postura parece não ser a mais adequada como a seguinte afirmação deixa concluir:

> "*(...) it would be preferable to devote more effort to specifying the system's functionality in the earliest stage of the process before any design decisions have been made, since such early effort could lead to large overall savings.*" [Gajski et al., 1994, pág. 10].

Os clientes são, normalmente, especialistas no domínio de aplicação do sistema, mas têm parcos conhecimentos na área da informática e, mais particularmente, na modelação de sistemas. Por outro lado, os projectistas do sistema são, por definição, especialistas no projecto de sistemas computacionais e, muito provavelmente, têm conhecimentos superficiais, na área de aplicação. Estes conhecimentos diferentes provocam perspectivas diferentes quanto ao sistema, o que implica que a comunicação entre ambas as partes seja difícil, uma vez que, para além de não haver uma percepção das necessidades da outra parte, a própria terminologia usada é diferenciada. Neste sentido, as especificações são úteis às duas partes envolvidas no projecto dum sistema: para o cliente, porque assim garante que o seu problema foi correctamente entendido e para

[17] Trata-se duma solução de compromisso entre a legibilidade (simplicidade) do documento e a precisão da descrição da funcionalidade pretendida para o sistema.

[18] Numa especificação descrita por uma rede de Petri, por exemplo, estas três hipóteses são possíveis: inspecção visual, uso dum editor/simulador ou recurso a rotinas automáticas de validação de propriedades.

o projectista pois, desta forma, obtém uma ideia clara do problema, o que lhe permite iniciar imediatamente o desenvolvimento do produto [Calvez, 1993, pág. 189]. A especificação é, pois, um meio de comunicação entre os clientes e os projectistas, como a fig. 2.12 pretende ilustrar.

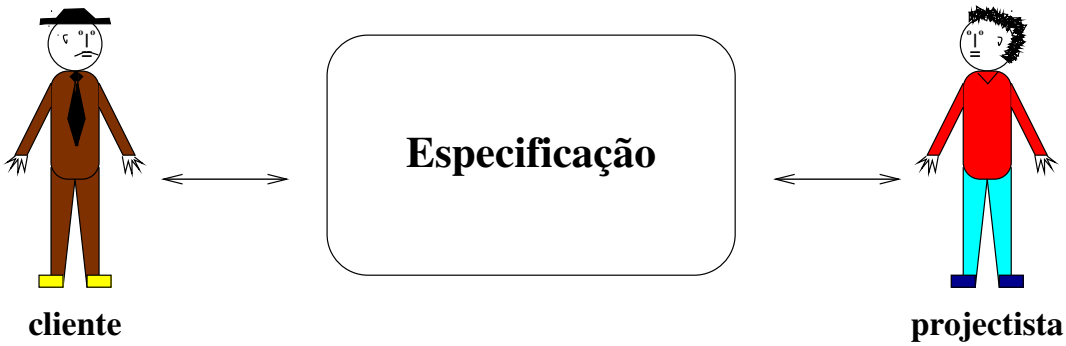


Figura 2.12: A especificação como meio de comunicação entre clientes e projectistas.

Para que as especificações possam funcionar como meio de comunicação privilegiado entre clientes e projectistas, o autor advoga a utilização de notações gráficas que sejam claras e intuitivas, de forma a que as especificações possam ser criadas, alteradas e inspeccionadas, tanto por clientes como por projectistas, mas, simultaneamente, precisas e rigorosas, a fim de serem usadas por computadores para validação, simulação ou análise. O uso de notações matemáticas pode também ser considerada uma solução possível se houver um mecanismo que automaticamente as converta numa linguagem entendida pelo cliente; caso contrário, a sua aplicabilidade reduzir-se-á, actualmente, a um conjunto muito restrito de projectos.

2.4.3 Meta-modelos de especificação

Os vários meta-modelos de especificação de sistemas dividem-se, genericamente, em 5 categorias diferentes, segundo a taxinomia apresentada por Gajski *et al.*[19]: (1) baseados em estados; (2) baseados em actividades; (3) baseados em estrutura; (4) baseados em dados; e (5) heterogéneos [Gajski et al., 1994, pág. 19]. Estas categorias reflectem as diferentes perspectivas que se pode ter dum sistema, nomeadamente a sua sequência de controlo, a sua funcionalidade, a sua estrutura ou os seus dados.

Um *meta-modelo baseado em estados* representa um sistema como um conjunto de estados ligados por transições, as quais são disparadas por eventos externos. Este tipo de meta-modelo é apropriado para especificar os aspectos de controlo (dinâmicos e temporais) dum sistema, sendo, portanto, de particular interesse para os sistemas reactivos. Exemplos deste tipo de meta-modelos são as máquinas de estados finitos [Clare, 1973], as redes de Petri [Murata, 1989], os state-charts [Harel, 1987] e os SpecCharts [Gajski et al., 1993]. O meta-modelo RdP-SI [Fernandes e Proença, 1994], que é uma extensão às Redes de Petri, desenvolvida no DI/UM, é um outro exemplo de meta-modelos baseados em estados. Este meta-modelo já foi experimentado, com sucesso, na especificação de controladores digitais paralelos [Fernandes et al., 1995] [Fernandes et al., 1997].

[19] O termo "modelo" tem, para Gajski *et al.*, um significado duplo: meta-modelo e modelo.

Um *meta-modelo baseado em actividades*, de que são um exemplo os diagramas de fluxo de dados (DFSs), descreve um sistema como um conjunto de processos (ou actividades) relacionados pelas dependências de dados ou de execução. Este meta-modelo aplica-se a sistemas transformacionais, como por exemplo, sistemas de processamento de sinal ou compiladores, onde os dados são processados à medida que vão passando de processo em processo.

Um *meta-modelo baseado em estrutura*, como os diagramas de blocos, representa os módulos dum sistema e respectivas interligações. Contrariamente aos meta-modelos baseado em estados e em actividades que reflectem sobretudo a funcionalidade do sistema, os meta-modelos baseados em estrutura abordam principalmente a composição física do sistema. Os esquemáticos, tradicionalmente usados em projectos de hardware, são outro exemplo emblemático deste tipo de meta-modelo.

Quando se usa um *meta-modelo baseado em dados*, o sistema é representado como uma colecção de dados relacionados pelos atributos, classe, etc. Os meta-modelos deste tipo são muito usados para desenvolvimento de sistemas de informação (exemplo, bases de dados), onde a perspectiva da organização dos dados do sistema é bem mais importante do que a própria função do sistema. Um tipo de meta-modelo baseado em dados é o diagrama entidade-relacionamento [Teorey, 1990], que define um sistema como um conjunto de entidades e as respectivas interligações. Outro exemplo deste tipo de meta-modelo é o diagrama de estrutura de Jackson [Davies e Layzell, 1993, cap. 6, pág. 105–13] que modela cada dado em termos da sua estrutura, decompondo-o, duma forma hierárquica, em sub-dados[20].

Um meta-modelo diz-se *heterogéneo*, se incorporar uma qualquer combinação das características dos quatro meta-modelos atrás descritos. São utilizados meta-modelos heterogéneos, quando se pretendem representar, no mesmo modelo, diversas perspectivas dum dado sistema. Como exemplo paradigmático dum meta-modelo heterogéneo tem-se os diagramas de fluxo de controlo e dados (CDFG), já que estes incorporam duas perspectivas diferentes numa única representação.

Os meta-modelos orientados ao objecto (cap. 3), que podem ser vistos, numa perspectiva histórica, como uma evolução ou extensão dos meta-modelos baseado em dados, são *meta-modelos com múltiplas vistas*, pois usam vários meta-modelos em simultâneo, de modo a representar diferentes perspectivas (vistas) do mesmo sistema. A metodologia OMT, por exemplo, aborda, na fase de análise, os três seguintes aspectos dos sistemas, usando para cada um deles um modelo distinto: a estrutura estática (modelo de objectos), a sequência de interacções (modelo dinâmico) e as transformações de dados (modelo funcional) [Rumbaugh et al., 1991, pág. 149]. Por seu lado, o ambiente STATEMATE [Harel et al., 1990], que é um conjunto de ferramentas, com características gráficas, que permite a especificação, a análise, a concepção e a documentação de sistemas reactivos de elevada complexidade, incorpora igualmente 3 meta-modelos diferentes, um para cada vista do sistema: os *module-charts* para indicar a vista estrutural do sistema (os seus componentes e respectivas interligações), os *activity-charts* para modelar a vista funcional do sistema (os seus processos) e os *state-charts* para especificar a componente de controlo do sistema (o seu comportamento ao longo do tempo).

O meta-modelo RdP-shobi [Machado et al., 1997a], uma outra extensão às Redes de Petri, também desenvolvido no DI/UM, pode igualmente ser considerado um meta-modelo com múltiplas vistas, na medida em que usa Redes de Petri para modelar a vista de controlo dos sistema e uma notação textual idêntica a uma linguagem orientada ao objecto para modelar os dados e os processos do sistema. Este meta-modelo foi utilizado na especificação de contro-

[20]Este diagrama pode igualmente ser usado para descrever as acções dum programa, pelo que pode também ser considerado um meta-modelo baseado em actividades.

ladores digitais e respectivo sistema controlado, em áreas tão diversas como o controlo industrial [Machado et al., 1997c], as interfaces de comunicações [Machado et al., 1998a] e a microarquitectura de processadores [Machado et al., 1998b].

Dado que não se considerou relevante, neste contexto, fazer uma apresentação aprofundada dos vários meta-modelos supracitados, pode ser encontrada mais informação nas referências indicadas ou no livro [Budgen, 1994, cap. 6].

Note-se que os meta-modelos compostos, abordando várias vistas, não se podem considerar heterogéneos, uma vez que a informação não é representada numa única notação. Cada vista usa uma notação própria e distinta e não há nenhuma relação formal entra as várias notações. Nos meta-modelos heterogéneos existe um único formato integrado para representar os sistemas, do qual se podem extrair as diferentes vistas daqueles.

É impossível captar todos os pormenores dum sistema complexo num só tipo de modelo [Booch, 1991, pág. 155], do que resulta a necessidade da utilização dum meta-modelo heterogéneo ou com múltiplas vistas. Para que as especificações sejam descritas ao nível do sistema, há, hoje em dia, um consenso alargado relativamente à necessidade de modelar o sistema de acordo com três vistas (dimensões) complementares: os objectos ou dados, os estados e as actividades [Calvez, 1996]. Cada vista descreve um aspecto do sistema, mas contém referências às outras, o que permite relacioná-las, ou seja, as vistas são complementares e relacionadas (fig. 2.13). Por exemplo, para caracterizar uma actividade do sistema, os dados manipulados têm de ser conhecidos e referidos e, portanto, definidos anteriormente.

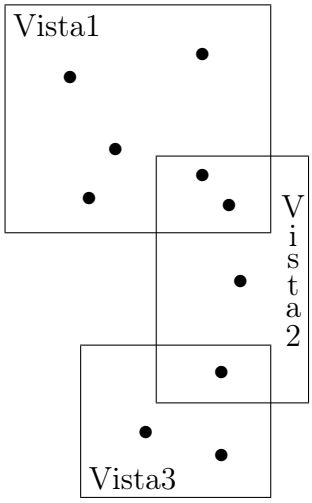


Figura 2.13: A relação entre as diferentes vistas dum sistema. Cada "•" representa um dado aspecto do sistema que foi considerado para modelação.

A vantagem de existirem disponíveis vários meta-modelos de especificação reside na possibilidade de se poderem representar diferentes vistas do mesmo sistema, fazendo assim acentuar as suas diversas características. Neste sentido, é manifestamente importante que a equipa de projecto escolha os meta-modelos apropriados para representar as características mais relevantes do sistema em desenvolvimento. Uma vez encontrados esses meta-modelos, o sistema pode ser especificado, indicando o seu funcionamento com exactidão e rigor.

2.5 Hardware e software

Nesta secção, pretende salientar-se as diferenças e as semelhanças entre o software e o hardware e mostrar que o projecto conjunto de sistemas a implementar em software e em hardware, abreviadamente referido por *co-projecto de hardware/software* (*Hardware/Software Co-design*), ou ainda mais simplesmente por *co-projecto*, é, nos tempos actuais, possível se forem utilizadas abordagens radicalmente diferentes das tradicionais (cf. fig. 1.4).

Nesta secção, pretende abordar-se mais pormenorizadamente e dum ponto de vista metodológico, de que forma o co-projecto pode beneficiar das inúmeras propostas feitas no âmbito da engenharia de software.

Assim, neste trabalho, assume-se o co-projecto como uma extensão da engenharia de software, mas tendo sempre presente a necessidade de enquadrar a sua utilização num plano mais alargado, de modo a que seja possível desenvolver, duma forma integrada, sistemas contendo, simultaneamente, software e hardware.

2.5.1 Semelhanças e diferenças

Como se viu na secção 2.2, os processos seguidos no desenvolvimento de hardware e de software são conceptualmente muito semelhantes, havendo um paralelo muito significativo nas fases de análise e de concepção. As diferenças mais significativas surgem apenas na fase de implementação, onde é necessário para o hardware criar objectos tangíveis o que não se verifica para o software.

Um sistema digital, seja ele implementado em hardware ou em software, pode ser descrito em diferentes domínios de representação: comportamental, estrutural e físico.

No domínio comportamental, indica-se o que o sistema faz e não a forma como ele é construído. O sistema é visto como uma "caixa negra" com entradas e saídas, definindo-se o comportamento do sistema através de relação das entradas e das saídas ao longo do tempo.

Uma representação no domínio estrutural ignora, na medida do possível, a funcionalidade do sistema e foca a estrutura do sistema em termos dos seus componentes e respectivas interligações. Embora a funcionalidade do sistema possa ser obtida, não é indicada explicitamente.

Uma representação no domínio físico é aquela que está mais perto da implementação final, indicando as características dos componentes usados na representação estrutural. Enquanto que a representação estrutural evidencia a ligação entre os componentes, a representação física descreve a relação espacial entre esses componentes, indicando o peso, o tamanho, o consumo e a posição de cada pino na implementação final.

Cada um destes três domínios pode ser subdividido em diversos níveis de abstracção (ou de granulosidade), diferenciando-se cada um deles pelo tipo de objectos que são usados nas especificações. Cada nível de abstracção representa um módulo de hardware a um dado nível de descrição.

Normalmente, um projecto de hardware percorre alguns dos vários níveis de abstracção a que um sistema digital pode ser visto. Tipicamente são considerados os seguintes níveis: Sistema (ou Arquitectural), Transferência de Registos (ou Micro-Arquitectural), Lógico e Dispositivo (ou Circuito).

O quadro 2.1 indica os elementos de descrição usados para cada nível de abstracção num dado

domínio de representação.

Níveis de abstracção	**Domínio de representação**		
	Comportamental	**Estrutural**	**Físico**
Sistema	Algoritmos	Processadores, Memórias	Circuitos, Macro-células
Transferência de Registos	Fluxo de dados	Registos, ALUs, MUXs	Disposição dos blocos sobre o silício
Lógico	Equações booleanas, Tabelas de verdade	Portas lógicas, Flip-flops	Células
Dispositivo	Equações eléctricas, Funções de transferência	Transístores, Condensadores, Resistências	Geometrias sobre o silício

Tabela 2.1: Níveis de abstracção e domínios de representação de sistemas digitais.

É importante notar que os níveis de abstracção dum sistema não são descrições de sistemas distintos, mas antes descrições do mesmo sistema a níveis de complexidade diferentes. Há, portanto, uma relação (de um para muitos) dos níveis de abstracção mais altos para os mais baixos, o que significa que uma descrição algorítmica, por exemplo, pode ser transformada em diferentes redes ao nível da transferência de registo. Os níveis mais abstractos são, por definição, mais independentes da implementação (quer isto dizer que há inúmeras implementações possíveis para o mesmo sistema) e mostram-se úteis quando ainda não se determinaram as tecnologias a usar.

Regra geral, é seguida, na descrição de sistemas hardware, uma abordagem descendente (do mais abstracto para o mais concreto), em que os níveis mais baixos são obtidos (sintetizados, na terminologia da área) a partir dos mais altos. Os níveis altos, por estarem menos comprometidos com uma solução final, possibilitam que o sistema seja simulado, mas são de difícil implementação directa. Desse modo, para implementar os sistemas descritos a um nível de abstracção alto são necessárias ferramentas de síntese, automáticas de preferência. A *síntese* é definida como o processo que transforma uma descrição comportamental numa descrição física [Gajski et al., 1992, pág. 8].

A prática mostra que o nível de abstracção mais baixo usado no software é o algorítmico, que é o mais alto usado em sistemas hardware. O nível do sistema ainda não conhece uma utilização vulgarizada, em projectos reais de hardware, devido sobretudo à inexistência de ferramentas de síntese, fiáveis e estabilizadas, que aceitem como entrada descrições a esse nível. Este problema não se verifica no software, pois o nível de automatização (e confiança) que os compiladores oferecem, fazem com que sejam grandes as diferenças entre a arquitectura do computador e as linguagens de programação de alto nível, razão pelo qual os projectistas recorrem, invariavelmente, a estas últimas.

Em termos de modelação, a manipulação da complexidade é crucial em ambos os domínios, pelo que o uso de mecanismos de abstracção se torna imperativo. A utilização de módulos e de componentes reutilizáveis é frequente em ambos os domínios, embora esteja mais sustentada nos projectos hardware [Oliveira, 1995], onde é comum o uso de componentes armazenados em bibliotecas. O hardware e o software partilham ainda uma série de características que

incluem a noção de funcionalidade, o conceito de estado e a possibilidade de programação [Kumar et al., 1996b, pág. 43].

Em software e, sobretudo, em hardware, além dos requisitos funcionais do sistema, que definem quais as funcionalidades que se pretendem para o sistema, é também necessário recolher uma série de outros requisitos (por exemplo, o preço, o desempenho, a fiabilidade e o tamanho), que irão condicionar as soluções de implementação para o sistema em desenvolvimento. Morris *et al.* sugerem a divisão dos vários requisitos dum sistema, segundo duas categorias: *requisitos funcionais* e *restrições de concepção* [Morris et al., 1996, pág. 96]. Estas últimas podem ainda ser especializadas em três sub-categorias: requisitos não funcionais, objectivos de concepção e decisões de concepção (fig. 2.14).

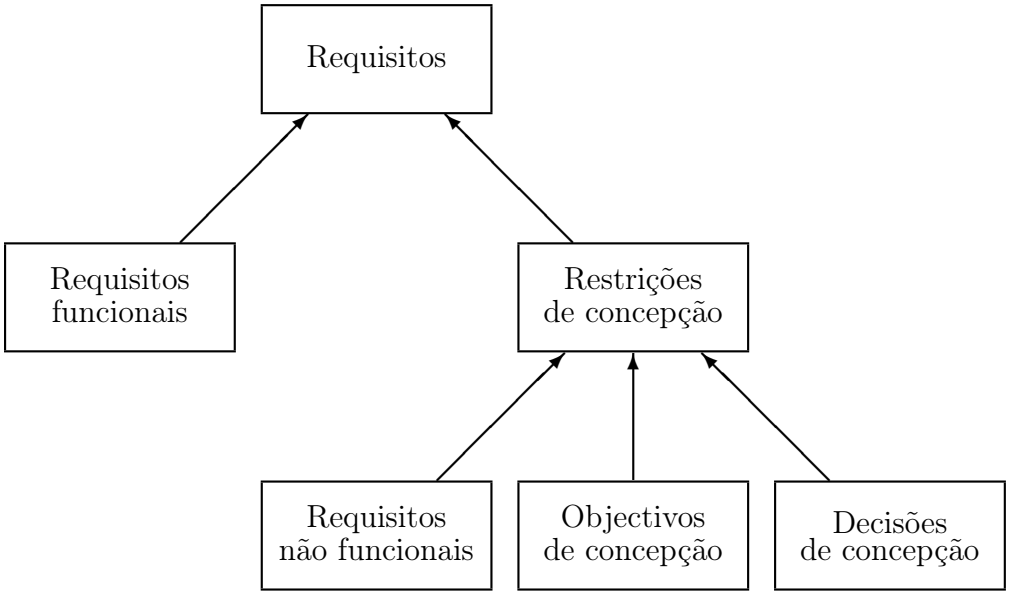


Figura 2.14: A categorização dos requisitos.

Os *requisitos não funcionais* especificam um conjunto de valores para grandezas como, por exemplo, o desempenho, o tempo de resposta, a fiabilidade, o custo ou as dimensões do sistema. Estes requisitos são muito úteis em fases mais adiantadas do desenvolvimento, uma vez que permitem, por exemplo, determinar quais as partes do sistema a implementar em hardware e em software.

Os *objectivos de concepção* estão normalmente relacionados com questões relacionadas com o teste, a manutenção e outros requisitos que não podem ser quantificados de forma precisa. Habitualmente, são indicados no enunciado do problema usando expressões como as seguintes: "tem de ser o mais rápido possível", "tem de ser barato", "tem que ser fácil de modificar". Cabe à equipa de projecto, sempre que possível, transformar este tipo de objectivos em requisitos não funcionais (por exemplo, definindo um valor máximo para o preço ou um nível de desempenho mínimo). Se tal não for possível, devem ser usados como critérios de selecção, quando houver alternativas funcionalmente equivalentes e não existirem outros critérios mais sólidos.

As *decisões de concepção* presentes no enunciado do sistema surgem normalmente quando o cliente tem alguns conhecimentos de técnicas computacionais e são usualmente consideradas indesejáveis, pelo que uma recomendação genérica é diligenciar a sua eliminação. Contudo, nem sempre é possível ignorar as decisões de concepção, pois, por exemplo, o sistema pode fazer

parte duma família de produtos (sendo necessário cumprir algumas directivas do fabricante para manter a linha da família) ou ser obrigatório incluir um barramento específico para ligação externa (exemplo, SCSI ou PCI). As decisões de concepção, por poderem influenciar tanto a funcionalidade a disponibilizar como apenas as decisões de implementação, são relevantes em diversas etapas do processo de desenvolvimento.

No desenvolvimento de hardware, a concepção está, como no desenvolvimento de software, dividida em duas actividades principais: a concepção arquitectural e a concepção detalhada. Na primeira, determinam-se os módulos necessários para implementar o sistema bem como o interface com o exterior daqueles e identificam-se os componentes padronizados a usar. Na concepção detalhada, para cada um dos módulos, desenvolve-se a sua estruturação interna. Há portanto uma semelhança enorme nesta fase, para o desenvolvimento de software ou de hardware.

A fase de implementação em hardware, além da tarefa de codificação que também existe em software, tem adicionalmente uma tarefa de fabricação, em que se realizam as implementações físicas que vão constituir o hardware do sistema em desenvolvimento. Durante a tarefa de fabricação, é necessário, por exemplo, construir placas de circuito impresso, configurar/programar os componentes do sistema e soldar/colocar na placa os circuitos integrados e outros componentes electrónicos (digitais e analógicos).

A principal diferença entre o hardware e o software reside no facto daquele ter uma existência tangível, a que estão associadas algumas características físicas, tais como, potência consumida, área ocupada e atrasos temporais. Não há, portanto, no software o equivalente à fabricação do hardware, i.e. à realização num substracto físico da funcionalidade do sistema. As características físicas tornam o desenvolvimento de hardware mais complicado, devido a questões como temporizações, ruídos, dissipação de calor e efeitos electromagnéticos que, em muitos casos, fazem com que o sistema tenha um comportamento completamente inesperado.

Finalmente, o hardware deve ser testado e, a exemplo do que sucede no software, são várias as actividades de teste existentes. Pelas razões indicadas no parágrafo anterior, o teste de sistemas hardware é, regra geral, mais complicado que aquele realizado em sistemas software.

2.5.2 Co-projecto de hardware/software

O *co-projecto*[21] é definido como o desenvolvimento integrado de sistemas digitais, implementados com componentes de hardware e de software [Buchenrieder e Rozenblit, 1995]. O co-projecto é uma abordagem unificada e cooperativa para o desenvolvimento de sistemas hardware/software, em que as opções de hardware e software podem ser consideradas em conjunto [Kumar et al., 1996b, pág. 5]. O co-projecto junta conceitos, ideias e técnicas de 3 disciplinas ligadas ao desenvolvimento de sistemas: modelação ao nível do sistema, desenvolvimento de software e desenvolvimento de hardware.

O tipo de sistemas que o co-projecto pretende desenvolver não é pois novo; trata-se de sistemas embebidos implementados com hardware e software. O que o co-projecto apresenta de diferente é a abordagem ou, duma forma mais lata, a metodologia, que pretende facilitar o desenvolvimento integrado desses sistemas. O co-projecto distingue-se do projecto tradicional, no sentido em que relaciona continuamente o desenvolvimento do hardware com o do software,

[21] O prefixo "co" tem diversas interpretações possíveis, podendo significar, entre outros, conjunto, coordenado, concorrente ou cooperativo.

e *vice-versa*. No co-projecto, o sistema é tratado como um só, e não como sendo constituído por duas componentes, desenvolvidas de forma separada após a sua separação inicial, que há que interligar, na fase final do projecto.

O interesse suscitado, na comunidade científica e na indústria, pelo co-projecto deve-se a variados motivos. Em primeiro lugar, os avanços em ambientes de especificação e simulação ao nível do sistema, em técnicas de prototipagem, em métodos formais para concepção e verificação, e em algoritmos para síntese de alto nível abriram novas possibilidades para o co-projecto. Por exemplo, os progressos registados, nos últimos anos, na síntese de alto nível e no desenvolvimento de ASICs (*Application Specific Integrated Circuit*) possibilitam que algoritmos complexos possam ser implementados em silício, rapidamente e a preços reduzidos. Por outro lado, a existência de processadores RISC (*Reduced Instruction Set Computer*), cada vez mais rápidos, permite transferir para o software, rotinas que anteriormente, por questões de desempenho, tinham de ser executadas em hardware.

Em segundo lugar, a complexidade crescente das aplicações que recorrem a sistemas embebidos requer metodologias de desenvolvimento avançadas e flexíveis. Em último lugar, é factor determinante, num mercado tão aberto e competitivo, como o dos sistemas embebidos, que o tempo e os custos de desenvolvimento e teste sejam cada vez mais reduzidos.

A decisão, manual ou automática, em colocar certos componentes do sistema em hardware e outros em software, tarefa conhecida habitualmente pelo nome de *partição hardware/software*, a par da escolha da representação a usar, da co-síntese (síntese do software, do hardware e das interfaces entre ambos) ou da co-simulação (simulação conjunta do hardware e do software), constitui um dos principais problemas que o co-projecto aborda, e baseia-se em factores tais como custo, velocidade, fiabilidade, ou frequência esperada de alterações.

As abordagens de partição podem ser classificadas de acordo com 4 características principais [Lavagno et al., 1996]:

- **Modelo de representação**: Podem ser grafos baseados na especificação ou notações matemáticas.
- **Nível de granulosidade** (que define as unidades de partição): Podem ser instruções elementares (de linguagens) ou blocos compostos (como processos, tarefas, ou ciclos).
- **Função de custo**: Pode basear-se, por exemplo, numa classificação do tipo de operações ou em estatísticas de custo e desempenho.
- **Algoritmo**: Podem utilizar-se heurísticas, métodos de agregação, melhorias graduais ou programação matemática.

Mais informações sobre partição de sistemas, e dum tópico estritamente relacionado, a afectação, podem ser encontradas em [Gajski et al., 1994, cap. 6]. A *afectação* (*allocation*) pode definir-se, neste contexto, como sendo a escolha dos componentes que vão permitir a implementação do sistema. Com a escolha dos componentes a usar definida, podem então partir-se a funcionalidade do sistema por aqueles.

No projecto tradicional, a partição do sistema é realizada no início do desenvolvimento, ao passo que no co-projecto essa tarefa só é realizada bem mais tarde. A fig. 2.15 pretende realçar essa importante diferença. Deste modo, o co-projecto dum sistema é possível, se a sua concepção for independente da representação final para implementação e antes de decidir a partição dos componentes para soluções em hardware ou em software. Como consequência, os métodos e as notações a usar devem representar o sistema e permitir que os seus componentes possam ser implementados tanto em hardware como em software. No caso do desenvolvimento de sistemas embebidos, o uso duma *representação unificada* para especificar simultaneamente as

componentes hardware e software dos sistemas é desejável [Kumar et al., 1996b, pág. 48]. Esta representação parte do pressuposto que o hardware e o software são equivalentes e que não há nenhuma diferença fundamental entre eles, como se pode concluir da seguinte afirmação:

> "*Hardware and Software are logically equivalent.*"
> [Tanenbaum, 1976, pág. 10] [Tanenbaum, 1999, pág. 8].

A ideia subjacente a esta afirmação é a de que qualquer operação realizada em software pode ser directamente construída em hardware e que, similarmente, uma instrução executada em hardware pode ser transferida para software.

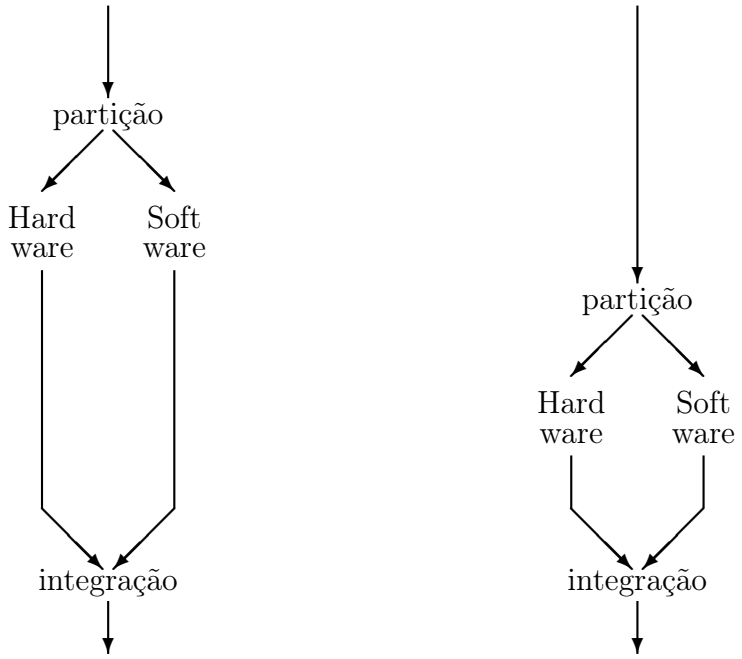


Figura 2.15: A partição no projecto tradicional e no co-projecto.

A utilização duma representação unificada e, consequentemente, duma partição tardia do sistema traz inúmeras vantagens. Em primeiro lugar, a obrigatoriedade em separar o desenvolvimento das componentes software e hardware deixa de ser inevitável (cf. fig. 1.4), podendo antes o uso de técnicas comuns de análise, de concepção e de teste ser adoptado. Uma representação unificada, além de disponibilizar uma plataforma comum para o desenvolvimento de hardware e de software, e de possibilitar a utilização de várias técnicas duma maneira uniforme, permite ainda a *fertilização cruzada* dos domínios hardware e software, ou seja, facilita que técnicas e resultados dum domínio possam ser aplicados ao outro [Smith e Gross, 1986]. Por exemplo, a simulação dum sistema com hardware e software (tarefa conhecida por *co-simulação*) num ambiente comum é, neste sentido, perfeitamente exequível. A exploração de compromissos hardware/software também pode ser realizada, uma vez que a transferência de funcionalidades entre o hardware e o software é facilitada [Kumar et al., 1996b, pág. 6].

Como se verá no cap. 3, os meta-modelos usados nas metodologias orientadas ao objecto apresentam um conjunto significativo de características, que permitem equacionar a sua utilização

como uma possível representação unificada para o desenvolvimento de sistemas embebidos[22]. Um dos objectivos deste trabalho encaixa precisamente nesta linha de investigação e consiste no estudo da adequabilidade dum meta-modelo orientado ao objecto, na área aplicacional dos sistemas embebidos.

2.6 Resumo final

Este capítulo iniciou-se com a apresentação da forma tradicional como o projecto dum sistema é seguido, indicando alguns dos problemas que esta abordagem implica. De seguida, fez-se um levantamento dos modelos de processo usados para desenvolvimento de sistemas, indicando as vantagens e desvantagens de cada um deles. Foram apresentadas as características mais relevantes das metodologias estruturadas e orientadas ao objecto e referiu-se em que sentido se podem considerar estas como mais adequadas, do que as primeiras, para modelar sistemas complexos.

Fez-se uma descrição das várias questões associadas à análise de sistemas e tentou-se demonstrar a necessidade desta fase não ser tratada superficialmente, no âmbito do desenvolvimento dum sistema complexo. Foram indicadas, para as especificações, as características desejáveis, com particular incidência na necessidade da especificação dum sistema ser formal, executável e ter associada uma notação gráfica que possa ser entendida por clientes e projectistas. Apontaram-se ainda as 5 categorias em que os meta-modelos de especificação se dividem e concluiu-se da necessidade de serem usados meta-modelos heterogéneos ou com múltiplas vistas para modelar sistemas complexos.

A finalizar, mostrou-se ainda que, à parte alguns pormenores, a forma seguida para desenvolver software é muito semelhante àquela adoptada para desenvolvimento de hardware e que é possível definirem-se representações unificadas para descrever tanto hardware como software, uma vez que estes são logicamente equivalentes. Este facto permite que o co-projecto de hardware/software possa ser uma realidade, sendo defendida, neste trabalho, a tese que o paradigma dos objectos, tradicionalmente usado para sistemas software, pode ser igualmente utilizado para desenvolver de forma integrada sistemas hardware/software.

[22]Por exemplo, a utilização da metodologia OMT para hardware já foi experimentada por diversas equipas de investigação [Morris et al., 1996] [Mariatos et al., 1996] [Leblanc, 1996] [Schumacher et al., 1996]. Por outro lado, uma série de equipas de investigação [Harel e Gery, 1997] [de la Puente et al., 1998] [Maffezzoni et al., 1998] advoga que o uso do paradigma do objecto e das técnicas da engenharia de software, no contexto do desenvolvimento dos sistemas hardware e, em particular, dos sistemas embebidos, pode constituir uma abordagem válida e eficaz.

Capítulo 3

Modelação de Sistemas com Objectos

Tal pai, tal filho.

Quem sai aos seus não degenera.

Sumário

Este capítulo é iniciado por uma caracterização genérica e informal dos objectos, que é seguida por uma definição do termo objecto, no contexto deste trabalho. Faz-se ainda uma referência ao significado do termo "orientado ao objecto", relativamente aos termos "baseado em classes" e "baseado em objectos". São depois indicados vários conceitos e características associados à orientação ao objecto, que permitem mostrar os mecanismos de modelação que o projectista tem à sua disposição para lidar com a complexidade dos sistemas. De seguida, é feita uma apresentação da notação UML, usada para modelar, segundo a abordagem dos objectos, as várias vistas dum sistema complexo e o capítulo é concluído com um levantamento das questões relacionadas com os utilitários que têm de existir para o desenvolvimento orientado ao objecto ser exequível.

Índice

3.1 Âmbito do termo "objecto"

Pretende, nesta secção, caracterizar-se qual o significado do termo objecto no contexto deste trabalho e dar uma definição textual o mais completa desse termo. Pretende ainda demonstrar-se que um objecto, quando usado para modelar parte dum sistema, pode ser visto segundo várias alternativas, ou seja, que um objecto congrega, em si mesmo, várias perspectivas de modelação. A finalizar, discute-se o significado do termo "orientado ao objecto" no âmbito da modelação de sistemas.

3.1.1 Caracterização dos objectos

A orientação ao objecto é uma técnica para modelação de sistemas, que faculta, para tal, um conjunto significativo de conceitos e mecanismos especialmente apropriados a esse fim. Quando se recorre ao paradigma dos objectos para modelar um dado sistema, passa-se a olhar para este como um conjunto de objectos que interagem entre si. Uma vez que, no dia-a-dia, as pessoas também lidam com objectos, é relativamente simples pensar de forma semelhante quando se pretende construir um modelo. Assim, o conceito de *objecto* passa a ser fundamental nesta abordagem.

Como exemplo, para auxiliar a caracterização dos conceitos mais importantes dos objectos, considere-se um sistema composto por pessoas que desenvolvem certas actividades. Far-se-á uma modelação desse sistema à luz da abordagem orientada ao objecto, tentando evidenciar o quão natural resulta a construção desse modelo. A escolha dum exemplo não directamente relacionado com a Informática tem por objectivo demonstrar que o paradigma dos objectos se adapta igualmente bem a qualquer domínio de aplicação.

Um sistema, no âmbito da modelação por objectos, é composto por vários objectos, que correspondem a entidades reais. O exemplo considerado inclui, entre outras pessoas, o Joaquim e a Madalena, que são vistos como dois objectos do sistema em questão.

Um objecto contém um conjunto de dados (informação), que se designam *atributos*, que o descrevem de alguma forma. Para o Joaquim, além do seu nome, pode acrescentar-se a sua idade, a sua morada, o seu sexo, a sua profissão, etc. Para manipular estes atributos, devem definir-se as *operações* que afectam ou permitem usar aqueles. A única parte do objecto visível do exterior são as suas operações, mais propriamente as respectivas interfaces, estando todo o resto escondido (quer os atributos, quer a forma como as operações são implementadas). Para saber como estas são executadas, há que "olhar" para dentro do objecto. Na fig. 3.1 mostra-se o objecto Joaquim e os seus atributos e as suas operações, usando uma notação intuitiva, mas sem qualquer semântica definida.

A relação ou associação entre os objectos também tem de ser modelada. Pode ser importante modelar que o Joaquim telefona à Madalena ou que o Joaquim é casado com a Madalena. A primeira é uma *relação dinâmica*, já que permite que os objectos comuniquem (i.e. troquem informações entre si) e a segunda é uma *relação estática*, pois tem uma duração mais longa e implica que cada objecto sabe da existência do outro. Podem existir outros tipos de relação entre os objectos, nomeadamente a *composição* ou a *agregação*. O Joaquim pode ser visto como sendo estruturalmente constituído pela cabeça, braços, tronco e pernas. A outro nível, pode alternativamente indicar-se que o Joaquim e a Madalena são casados, criando-se um objecto casal, que agrega os dois objectos que o compõem.

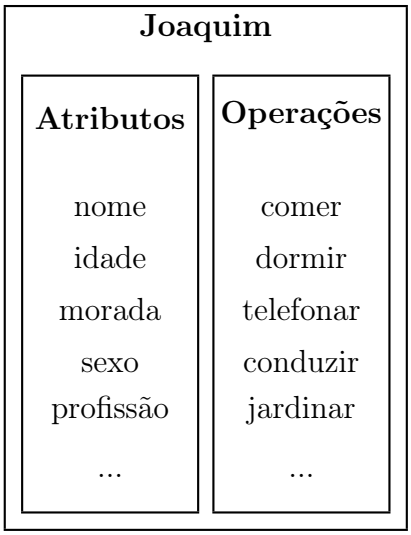


Figura 3.1: O objecto Joaquim, caracterizado pelos seus atributos e operações.

O comportamento temporal do modelo dos objectos é conseguido, devido às relações dinâmicas entre objectos, pois estas permitem que os objectos enviem estímulos (mensagens, na terminologia da orientação ao objecto) uns aos outros. Um objecto, ao receber uma mensagem, realiza uma determinada operação e pode provocar novas mensagens em outros objectos. Se o Joaquim pedir à Madalena para cozinhar, esta pode, por sua vez, solicitar à sua filha para ir às compras, de modo a poder satisfazer o pedido inicial. Os objectos só respondem às mensagens para as quais estão preparados. Se o Joaquim pedir à Madalena para trocar o pneu furado do carro, esta pode responder-lhe que não sabe realizar tal operação; assim, o Joaquim terá de tornar a fazer o pedido, mas desta vez ao seu mecânico para ter a certeza que o pneu será efectivamente trocado. Este exemplo mostra que os pedidos que são feitos a um objecto podem ou não ser satisfeitos, dependendo este facto das operações que aquele está habilitado a desempenhar.

Depois desta apresentação sucinta dos sistemas orientados aos objectos, far-se-á, na secção seguinte, uma descrição mais completa do termo objecto no contexto da modelação de sistemas.

3.1.2 Definição de objecto

O termo "*objecto*" é sinónimo de "*coisa*", razão pela qual se procedeu à consulta de enciclopédias, de que resultou a recolha das seguintes definições para esses dois termos:

> **Objecto**: Tudo aquilo que se apresenta aos sentidos; aquilo que se apresenta à vista. (...) Em sentido geral, qualquer coisa; peça, artigo (...).
> [Grande Enciclopédia Portuguesa e Brasileira, 1975, vol. 19, pág. 111].

> **Cousa**: Em sentido geral, tudo quanto existe (...). Em sentido mais restrito, designa todo ser, real ou imaginário, tudo o que se diz ou pode dizer, tudo o que se faz ou pode fazer (...).
> [Grande Enciclopédia Portuguesa e Brasileira, 1975, vol. 7, pág. 942].

> **Objecto**: Do latim *objectum*, do verbo *objicere* (lançar ou pôr em frente, propor, expor, opor). Etimologicamente, objecto é o que está à frente a, o que se opõe ao

> sujeito. Por extensão, significa o que é, as coisas. Palavra do vocabulário comum que se restringe em sentidos muito precisos nas diversas disciplinas científicas e filosóficas (...).
> [Enciclopédia Luso-Brasileira de Cultura, 1973, vol. 14, pág. 426].

O termo *objecto* tem um significado, de uso genérico, estabelecido há muito tempo, pelo que será dada, neste capítulo, uma definição mais precisa, e de acordo com os princípios da modelação de sistemas computacionais, para dissipar qualquer interpretação de índole mais pessoal que se possa eventualmente fazer.

A última definição aponta para a necessidade em se encontrar uma definição mais restrita, uma vez que as definições dadas são, em certo sentido, bastante genéricas. O conceito de objecto, no projecto de software, surgiu, pela primeira vez, num construtor da linguagem de programação SIMULA, nos anos 60. Mais recentemente, este conceito serviu de base a diversas linguagens de programação orientadas ao objecto (por exemplo, SMALLTALK, OBJECTIVEC, EIFFEL, C++ e JAVA), bem como a outras linguagens não orientadas ao objecto (por exemplo, ADA e MODULA-2). Diversos métodos de análise e concepção, como referido no cap. 2, fizeram também uso, como elemento basilar, do conceito de objecto. Esta evolução determinou que o conceito de objecto fosse evoluindo e alargando o seu âmbito.

Um *objecto* pode ser visto como uma entidade, cujo comportamento se caracteriza pelo conjunto de acções que executa e que requer a outros objectos, podendo sugerir-se a seguinte definição:

> "*An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class*". [Booch, 1991, pág. 77].

Alternativamente, um objecto é simplesmente algo que tem de fazer sentido no contexto da aplicação, como a seguinte afirmação atesta:

> "*We defined an object as a concept, abstraction or thing with crisp boundaries and meaning for the problem at hand. Objects serve two purposes: They promote understanding of the real world and provide a practical basis for computer implementation. Decomposition of a problem into objects depends on judgment and the nature of the problem*". [Rumbaugh et al., 1991, pág. 21].

No âmbito da modelação de sistemas, a seguinte definição parece ser a mais apropriada, i.e., mais específica para a área em que este trabalho se enquadra:

> "*An object represents an individual, identifiable item, unit or entity, either real or abstract, with a well defined role in the problem domain. An object may be recognized by the data it carries, by its behavior (i.e. how it responds to events in its environment), and by the processing that it performs*". [Smith e Tockey, 1988].

Uma vez aceite esta última definição, um objecto, no contexto do desenvolvimento de sistemas complexos, apresenta o seguinte conjunto de características [Robinson, 1992, pág. 34]:

- Encapsula informação.
- Indica o tipo dos seus atributos.
- Disponibiliza operações a outros objectos.
- Pode requerer e usar operações disponibilizadas por outros objectos.

- Tem visibilidade reduzida dos outros objectos.
- Pode decompor-se noutros (sub-)objectos que, conjuntamente, exibem o mesmo comportamento.
- É um modelo duma entidade real ou uma entidade relevante para a solução final em software[1].
- É referido pelo seu nome.
- Pode ser uma instância duma classe.
- Pode implementar-se com um tipo abstracto de dados[2].
- Pode ter estado, ou seja, pode ser visto como uma máquina de estados.

Este conjunto de características pode ser resumido na seguinte definição:

> "*An object is a model of a real-world entity or a software solution entity that combines data and operations in such a way that data are encapsulated in the object and are accessed through the operations. An object thus provides operations for other objects, and may in turn also require operations of another object. An object may have state, either explicitly to provide control or implicitly in terms of the value of the internal data. An object may be a class or an instance created as a parameterisation of a class.*" [Robinson, 1992, pág. 34].

A visão de que os objectos podem ser vistos como tipos abstractos de dados, apesar de correcta, é demasiado restrita ou limitadora relativamente à complexidade dos sistemas que se pretende desenvolver. Não se apadrinha aqui essa única perspectiva de entender os objectos, uma vez que tem associado um nível de granulosidade muito fino, impedindo assim que alguns componentes mais complexos de certos sistemas possam ser vistos como objectos.

A exemplo do que fizeram outros autores [Selic et al., 1994, pág. 49–51], pode então tomar-se uma perspectiva distinta, mas não contraditória: um objecto é visto como uma máquina (agente computacional), no sentido de ser um componente dum sistema computacional. Esta perspectiva não é contraditória com o conceito de tipo abstracto de dados, pois um agente pode encapsular dados e disponibilizar um conjunto de métodos a aplicar a esses dados. Contudo, o conceito de máquina (ou agente) é mais genérico e abrangente e não tem as conotações habituais dos tipos abstractos de dados: âmbito restrito (grão fino), centrado nos dados e comportamento passivo. É simples entender uma máquina a um nível de granulosidade grosso, já que um sistema completo pode ser visto como uma única máquina.

O comportamento dum agente é, por inerência, mais complexo que o dum tipo abstracto de dados. Uma instância dum tipo abstracto de dados está sempre num estado passivo e "acorda" sempre que uma das suas operações é invocada. Assim que esta for concluída e o eventual resultado devolvido, a instância volta à situação de passividade, até nova invocação. Portanto, uma instância está activa durante as interacções com os seus clientes. Ao contrário, um agente computacional, por ter o seu próprio fio de controlo (*control thread*), pode estar activo durante um largo período de tempo, mesmo não sendo invocada qualquer das suas operações.

Programar com objectos consiste em construir algumas máquinas e interligá-las adequadamente, de forma a conseguir a funcionalidade pretendida. É um processo idêntico ao que os engenheiros de hardware seguem para construir sistemas mais complexos, a partir de componentes padronizados.

[1] Robinson considerou apenas o software, mas parece pacífico alargar o âmbito das características, de modo a perspectivar também os objectos como entidades de hardware.

[2] Um tipo abstracto de dados define os dados e as operações que podem operar sobre esses dados.

3.1.3 As várias perspectivas dum objecto

A definição de objecto dada por Smith e Tockey (pág. 66) indica que um objecto inclui, em simultâneo, as três perspectivas fundamentais sob as quais um sistema é, actualmente, modelado: os dados que incorpora, o comportamento dinâmico que exibe e os processos que realiza. A fig. 3.2 pretende ilustrar esse facto.

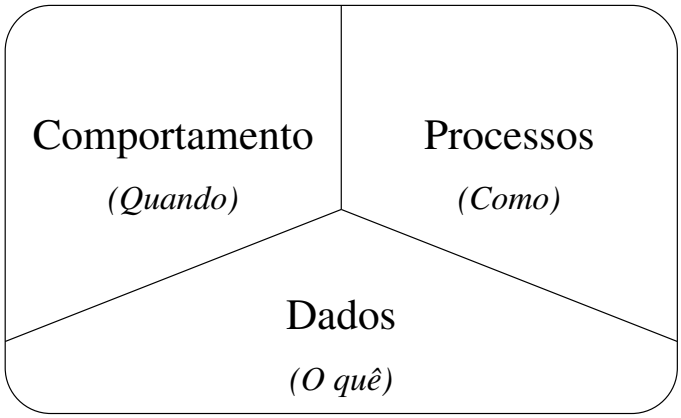


Figura 3.2: Os objectos caracterizados pelos dados, comportamento e processos.

Usando como exemplo um objecto do tipo impressora, a fig. 3.3 mostra como esses aspectos se relacionam. O conjunto de setas mostra que a actividade que consiste em colocar folhas no tabuleiro modifica a variável que indica se há papel disponível para impressão, e que tal só se pode realizar quando a impressora está num estado de pausa. Nem todos os objectos exibem os três aspectos num dado sistema, pelo que só aqueles que são relevantes para o domínio do problema é que devem ser modelados.

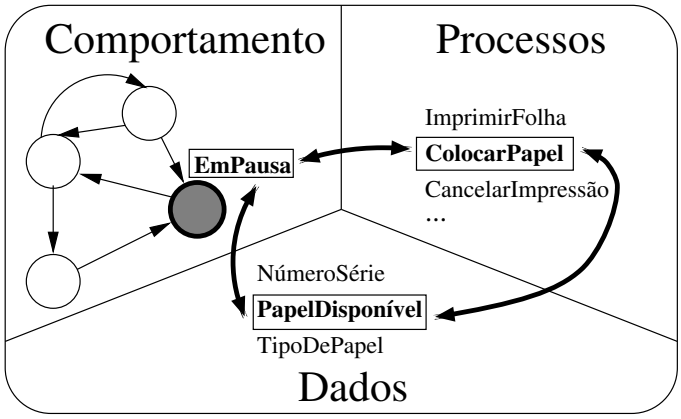


Figura 3.3: Um objecto impressora caracterizado pelos dados que manipula, o comportamento que exibe e os processos que realiza.

Pode, pois, afirmar-se que, para modelar um objecto (i.e. um sistema), há, regra geral, que modelar os seus três seguintes aspectos:

- **Dados** ou estrutura: Descrevem as características estáticas do objecto, nomeadamente os seus atributos relevantes para a aplicação em causa.
- **Controlo** ou comportamento global: Indica as dependências temporais e dinâmicas entre a ocorrência de eventos (externos) e a execução das acções internas, o que implica alterações no estado e nos dados do objecto.
- **Processos** ou comportamento local: Descrevem as actividades internas do objecto, em função dos estímulos externos a que está sujeito.

Os dados dum objecto são normalmente a primeira perspectiva a ser considerada para modelação, uma vez que, como se viu em capítulos anteriores, são considerados mais estáveis que as funções que realiza ou o comportamento que exibe.

3.1.4 Uso do termo "orientado ao objecto"

O termo "orientado ao objecto" tem sido usado, por vezes, de forma abusiva, uma vez que lhe está associada uma áurea de sucesso, pois é visto como sinónimo imediato de qualidade, modernidade, modularidade, reutilização ou robustez. Convém pois diferenciar claramente o significado dos termos "baseado em objectos", "baseado em classes" e "orientado ao objecto"[3]. Qualquer um destes termos pode ser usado para qualificar uma linguagem, um estilo de programação, o tipo de análise ou de concepção, uma metodologia, um sistema, uma aplicação, etc.

A programação baseada em objectos é definida como um estilo de programação que suporta encapsulamento e identidade de objectos. As operações e os atributos são escondidos numa cápsula (o objecto), a qual tem identidade própria. Não há suporte para o conceito de classe e, muito menos, para o mecanismo de herança entre classes. Ada [Ledgard, 1981] e, por arrastamento, VHDL [IEEE, 1994] são exemplos de linguagens baseadas em objectos.

As linguagens baseadas em classes incluem todas as características das linguagens baseadas em objectos e acrescentam o conceito de classe e as relações entre uma classe e as suas instâncias. No entanto, este tipo de linguagens não inclui ainda o conceito de herança. Entre as linguagens baseadas em classes inclui-se CLU [Liskov, 1993] [Guttag, 1993].

Os sistemas orientados ao objecto apresentam todas as propriedades dos sistemas baseados em objectos e em classes e adicionam a herança entre classes e a *recursividade própria*[4] (*self recursion*). As linguagens C++ [Stroustrup, 1991], Java [Flanagan, 1996] e SmallTalk [Goldberg e Robson, 1983] são exemplos de linguagens orientadas ao objecto, sendo mesmo esta última considerada a linguagem orientada ao objecto por excelência e, para o principiante no paradigma do objecto, a sua utilização é considerada como muito pedagógica, por permitir captar, duma forma muito límpida, a maioria dos conceitos ligados à temática [Coad e Yourdon, 1991, pág. 41].

Sumariando, as seguintes equações relacionam os três termos [Graham, 1991, pág. 28]:

baseado em objectos = encapsulamento + identidade

baseado em classes = baseado em objectos + classes

[3]Tratam-se de traduções livres dos termos em língua inglesa "object-based", "class-based" e "object-oriented", respectivamente.

[4]Recursividade própria refere-se à capacidade dum objecto poder chamar recursivamente os seus próprios métodos.

orientado ao objecto = baseado em classes + herança + recursividade própria

Wegner também aborda, duma maneira bem simplista mas elucidativa, esta questão, na óptica das linguagens [Wegner, 1990]. Em seu entender, as linguagens baseadas em objectos suportam a funcionalidade dos objectos mas não permitem a sua gestão. Por seu lado, as linguagens baseadas em classes permitem a gestão dos objectos, mas não incluem mecanismos para a gestão das classes. Finalmente, as linguagens orientadas ao objecto suportam a funcionalidade dos objectos, a gestão dos objectos pelas classes e a gestão destas através do mecanismo de herança. Resumindo, pode afirmar-se:

linguagens baseadas em objectos são aquelas que suportam objectos.

linguagens baseadas em classes são aquelas em que todos os objectos pertencem a uma dada classe.

linguagens orientadas ao objecto são aquelas em que as classes podem usufruir do mecanismo de herança.

3.2 Conceitos

Até ao momento, apenas se introduziu o conceito de objecto. No entanto, o paradigma dos objectos não se restringe apenas à existência de objectos e apresenta muitos mais conceitos, propriedades e características. Serão descritos, nesta secção, os conceitos mais importantes que caracterizam os objectos, bem como outros tópicos normalmente associados à temática da orientação ao objecto. Não será dada uma definição precisa e formal dos vários conceitos, já que pretende apenas fazer-se uma descrição relativamente completa desses conceitos e mostrar a sua relevância na modelação de sistemas complexos.

Não existe ainda um consenso na comunidade científica de quais as características que uma dada metodologia (ou em sentido mais estrito, o meta-modelo associado à respectiva notação) deve contemplar para ser considerada "orientada ao objecto". Rumbaugh *et al.* identificam as seguintes quatro como imprescindíveis: identidade, classificação, polimorfismo e herança [Rumbaugh et al., 1991, pág. 1]. Existem, para Booch, quatro elementos indispensáveis ao paradigma dos objectos, sem os quais um modelo não pode ser considerado orientado ao objecto: abstracção, encapsulamento, modularidade e hierarquia [Booch, 1991, pág. 38]. Existem ainda, segundo Booch, associados à orientação ao objecto outros três elementos, considerados importantes e úteis, mas não essenciais: tipagem, concorrência, e persistência.

Relativamente às linguagens, também aqui, o consenso ainda não foi obtido, relativamente às características que devem apresentar para serem rotuladas de orientadas ao objecto, como a seguir se pode constatar. Para alguns autores, as linguagens podem ser consideradas orientadas ao objecto se incluírem as seguintes características: encapsulamento, abstracção, ligação dinâmica e herança [Pascoe, 1986]. Para Jacobson *et al.*, as linguagens orientadas ao objecto devem incluir os seguintes pontos: objectos encapsulados, classes e respectivas instâncias, herança entre classes, e polimorfismo [Jacobson et al., 1992, pág. 84].

Estas divergências são reveladoras do estado pouco maduro que o paradigma dos objectos detém, o que significa que ainda são necessários mais alguns anos de investigação e experiência com projectos reais, até que se chegue a um consenso alargado sobre quais as características que as linguagens e os sistemas orientados ao objecto devem apresentar, para serem adjectivados como tal.

3.2.1 Identidade

A *identidade* significa que os dados do sistema podem ser quantificados em entidades discretas, a que se dá o nome de objectos. Cada *objecto* tem a sua própria identidade, o que significa que dois objectos são distintos, mesmo que tenham todos os seus atributos iguais. Por exemplo, duas bolas de ténis completamente iguais, da mesma cor, e do mesmo tamanho, têm existências distintas, pois cada uma delas existe *per si*. O termo identidade pressupõe que cada objecto é distinto pela sua própria existência e não pelo valor dos seus atributos.

Os objectos podem ser coisas reais (tangíveis), como maçãs, cães, televisores, impressoras, ou máquinas, mas também podem ser entidades puramente conceptuais como contas bancárias, casamentos, ou listas. Os objectos podem ainda ser entidades visuais, como letras, histogramas, gráficos ou círculos.

3.2.2 Classificação

A *classificação* significa que os objectos com a mesma estrutura de dados (atributos) e o mesmo comportamento (operações) são agrupados numa mesma classe. Um *atributo* é um elemento de modelação que descreve uma instância, caracterizando algo de importante e significativo acerca da natureza dessa instância. Exemplos de atributos, para uma classe conta bancária, incluem o nome do titular, a morada, o tipo de conta, a agência em que foi aberta e o saldo. Uma *operação* (ou *serviço*) é uma acção ou transformação desempenhada por um objecto ou a que o objecto se sujeita. Exemplos de operações, para a classe conta bancária, são abrir conta, fechar conta, alterar morada do titular, efectuar depósito e levantar numerário.

Uma *classe* (ou, dito duma forma mais precisa, uma classe de objectos) é uma abstracção que, para uma dada aplicação, define apenas as *propriedades*[5] mais relevantes e ignora as restantes. A escolha de quais as classes com relevância para uma determinada aplicação é arbitrária e depende fortemente desta.

Cada classe descreve um padrão que permite definir um conjunto, possivelmente infinito, de objectos, que partilham os mesmos atributos e operações [Graham, 1991]. O conceito de classe é algo semelhante ao de tipo de dados nas linguagens de programação mais tradicionais[6]. Um dado objecto diz-se instância duma classe, da mesma forma que uma variável é dum dado tipo. Uma classe inclui implicitamente a implementação de todos os objectos por ela definidos, dum modo que esconde os pormenores aos outros objectos e forçando os utilizadores dum determinado objecto a requerem os serviços deste através duma interface bem definida, a que se dá o nome de *assinatura* do objecto. Adicionalmente, a definição de classes pode levar à construção dum repositório de componentes reutilizáveis, o que facilitará o projectista em desenvolvimentos futuros.

Um objecto é uma *instância* duma classe, contém espaço próprio para os seus atributos (a que se chamam *variáveis de instância*) e o acesso a estes é unicamente possível usando as operações definidas na respectiva classe.

Em termos gráficos, segundo a notação UML (secção 3.3), um objecto distingue-se duma classe

[5] Usa-se o termo "propriedade" para referir indistintamente um atributo ou uma operação.

[6] Uma classe e um tipo não são exactamente o mesmo conceito, embora a distinção entre ambos não seja significativa em termos práticos. É suficiente afirmar que uma classe implementa um tipo [Booch, 1991, pág. 59]. Neste trabalho, o conceito de classe é usado para designar a especificação de objectos incluindo a implementação e a interface, enquanto que o conceito de tipo fica reservado para descrever apenas o domínio dos atributos.

da forma que a fig. 3.4 documenta. Uma classe é representada por um rectângulo, dividido em três secções: o seu nome é escrito, em negrito, na secção superior; os seus atributos e respectivos tipos são listados na secção interior; e as operações a que responde, bem como os eventuais parâmetros, são indicados na secção inferior. As duas últimas secções podem ser simplificadas (por exemplo, não indicando o tipo dos atributos ou os parâmetros das operações) ou mesmo omitidas, em função do nível de abstracção pretendido. Um objecto é igualmente representado por um rectângulo, onde se escreve, sublinhando, o nome da instância (opcional) e a classe a que pertence, bem como o valor dos atributos. Podem, igualmente, omitir-se alguns ou todos os valores dos atributos, quando tal informação não for tida por relevante no contexto em causa.

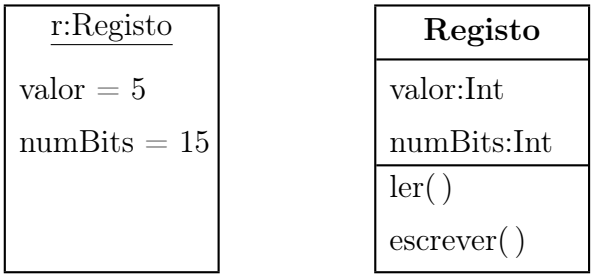


Figura 3.4: Notação gráfica para objectos e classes.

3.2.3 Abstracção

A *abstracção* consiste na selecção dos aspectos essenciais duma entidade, ignorando aqueles tidos por irrelevantes. O analista do sistema deve ter conhecimento, ou pelo menos saber da existência, dos vários aspectos duma dada entidade, para depois escolher unicamente aqueles que lhe pareçam importantes para o sistema a desenvolver.

A escolha de quais as propriedades essenciais depende fundamentalmente do propósito do sistema em causa. Aquilo que numa situação pode ser relevante, pode noutras ser completamente indiferente.

Uma abstracção indica as características essenciais dum objecto que o distinguem dos outros tipos de objectos e que, assim, marca a fronteira conceptual do objecto (i.e. os seus contornos), relativamente à perspectiva do analista.

Quando se usa abstracção, está a admitir-se que o objecto em causa é complexo. Não se pretende compreender e modelar esse sistema na sua totalidade, mas apenas escolher parte das suas características. Trata-se duma técnica muito importante para permitir lidar com a complexidade dos sistemas.

A abstracção foca a vista externa dum objecto e serve para separar o comportamento principal dum objecto da sua implementação. A passagem de mensagens usada para comunicação entre objectos facilita a abstracção, dado que o emissor da mensagem tem uma visão abstracta do receptor e este tem igualmente uma visão abstracta daquele [Wegner, 1990].

Na fase de análise, o uso de abstracção significa que se manipulam conceitos ao nível do domínio de aplicação, sem decidir, antes de conhecer profundamente o problema, por que tipo de concepção ou implementação se deve optar. Decidir qual o conjunto apropriado de abstracções, sejam elas algorítmicas ou de objectos, para uma determinada aplicação é o ponto fulcral da análise de sistemas.

3.2.4 Encapsulamento

Outra técnica útil para permitir lidar com a complexidade dos sistemas é o *encapsulamento* (a que também se chama *ocultação da informação*), que consiste em separar o comportamento externo dum objecto, a que têm acesso outros objectos, dos pormenores de implementação desse objecto, que são ocultados aos outros objectos.

Enquanto que a abstracção permite que o projectista se concentre naquilo que deve ser feito, o encapsulamento possibilita que se façam alterações num objecto, duma forma simples e sem interferência nos restantes objectos do sistema.

A abstracção e o encapsulamento são conceitos complementares, dado que enquanto a primeira foca na vista externa do objecto, o último evita que os clientes do objecto (aqueles que o usam) necessitem de conhecer a sua vista interior onde o comportamento da abstracção é indicado.

Para que a abstracção seja efectiva, as implementações têm que ser encapsuladas. Cada classe deve ser composta por duas partes: uma interface e uma implementação[7]. A interface duma classe descreve apenas a sua vista externa, tendo implicitamente associado o comportamento comum de todos os objectos dessa classe. A implementação duma classe inclui a representação da abstracção, bem como o mecanismo para atingir o comportamento pretendido. A divisão clara entre interface e implementação representa uma importante separação de conceitos. A interface permite expressar todos os pressupostos que os clientes dum objecto dessa classe podem fazer sobre este, enquanto que a implementação encapsula (esconde) certos pormenores, aos quais não interessa que os seus clientes tenham acesso.

O encapsulamento é o processo de esconder todas as propriedades dum objecto que não contribuem para a sua caracterização essencial. O encapsulamento não é, de modo algum, uma característica exclusiva dos sistemas orientados ao objecto[8]. Contudo, pelo facto de se combinarem, numa única entidade, o estado e o comportamento, o encapsulamento surge duma forma mais natural, simples e poderosa em sistemas orientados ao objecto se comparados com sistemas mais convencionais. Cox e Novobilski consideram mesmo que o encapsulamento é o princípio fundamental da programação orientada ao objecto [Cox e Novobilski, 1991, pág. 9].

O encapsulamento evita que sejam criadas interdependências num programa, que depois obrigariam a grandes reformulações, quando se pretendesse proceder a alterações, por pequenas que fossem, nos requisitos do sistema. Na prática, apenas as alterações feitas à interface da classe é que afectam os clientes duma sua instância. Visto de outra perspectiva, o encapsulamento permite o uso de funcionalidades sem existir a necessidade de conhecer os pormenores das suas implementações, pelo que alterações nestas não implicam modificações nas aplicações que as usam.

Um objecto define uma cápsula opaca relativamente aos dois sentidos sob os quais ele pode ser visto (de dentro para fora e no sentido inverso). A cápsula não só esconde a estrutura interna aos objectos externos, como também previne que componentes internos possam ter contacto directo com o exterior. Esta característica desejável para facilitar a reutilização, nem sempre está presente noutras abordagens. Por exemplo, muitas linguagens de programação (como Pascal ou Algol) possibilitam a ocultação da informação num só sentido: um sub-bloco tem acesso ao contexto dos blocos a que pertence, mas não *vice-versa*. Como resultado deste facto,

[7] Em VHDL, cada componente do sistema é também composto por duas partes: a entidade, que corresponde à interface, e a arquitectura, que indica o respectivo comportamento.

[8] Outras áreas onde o encapsulamento tem sido usado com enorme sucesso são, por exemplo, sistemas operativos e bibliotecas de funções.

os blocos aninhados só podem ser reutilizados no âmbito do contexto imposto pelos blocos a que pertencem, o que se revela um factor muito limitativo.

3.2.5 Mensagens

Uma acção é iniciada, em linguagens orientadas ao objecto, através da transmissão duma *mensagem* ao objecto responsável por essa acção. Se um objecto aceita uma mensagem que lhe é enviada, está a assumir o compromisso de desempenhar essa operação. A uma implementação duma operação por uma determinada classe dá-se o nome de *método*. Todos os objectos duma determinada classe, como resposta a uma mensagem do mesmo tipo, usam exactamente os mesmos métodos. Normalmente, a distinção entre operação e método não é relevante, pelo que é comum usarem-se esses termos indistintamente.

A forma dos objectos comunicarem entre si, representando no domínio do hardware os sinais e os protocolos entre componentes, consiste no envio de mensagens que efectuam a invocação de métodos aos objectos receptores. Uma mensagem indica apenas a operação a executar, deixando a responsabilidade de como o fazer para o objecto receptor da mensagem.

Os sistemas orientados ao objecto pressupõem um modelo de comunicação do tipo cliente/servidor. O *cliente* faz pedidos ao servidor para realizar serviços, enquanto que o *servidor* é responsável por disponibilizar um conjunto de serviços que podem ser requisitados pelos seus clientes. Este modelo de comunicação pressupõe uma dependência reduzida entre o cliente e o servidor, no sentido deste não precisar dum conhecimento prévio de quais os clientes que a ele podem recorrer para desempenhar uma operação em que é especialista. Os clientes, por seu lado, precisam apenas de conhecer a identidade (leia-se a localização) do servidor, a quem dirigem os seus pedidos. Se um pedido carece de resposta, o servidor simplesmente devolve o resultado ao emissor, cujo endereço pode ser obtido na mensagem original. Esta dependência reduzida entre servidor e cliente, e portanto entre objectos, permite que novos clientes sejam adicionados e que outros sejam eliminados, sem ser necessário proceder a qualquer alteração no servidor.

Outra das vantagens deste modelo de comunicação reside no facto de a ênfase ser colocado nos serviços que o servidor disponibiliza, em detrimento do modo como eles são realizados, o que evidencia, ainda mais, a perspectiva dos objectos serem abstracções funcionais que encapsulam informação.

O modelo cliente/servidor preconiza ainda um desenvolvimento guiado por responsabilidades, no sentido que incentiva o projectista a identificar os serviços por que cada classe de objectos se responsabiliza em fornecer aos outros objectos [Wirfs-Brock e Wilkerson, 1989]. A técnica dos cartões CRC (*Class-Responsability-Collaboration*) enfatiza a perspectiva de que as classes podem ser descritas com base nas suas responsabilidades, em detrimento da habitual indicação dos atributos e métodos [Beck e Cunningham, 1989]. Uma *responsabilidade* é, neste contexto, uma descrição dos propósitos ou dos papéis que uma dada classe tem relativamente ao sistema considerado. As responsabilidades dum objecto estão a um nível superior ao dos atributos e das operações. Estes fornecem os meios pelos quais as responsabilidades são cumpridas, mas não as definem. Por exemplo, um elevador pode ter os atributos e as operações indicados no quadro 3.1. As responsabilidades do elevador são transportar verticalmente pessoas e bens e são garantidas pelos atributos e operações que o elevador disponibiliza.

Uma mensagem passada do objecto emissor para o receptor pode ser vista como uma abstracção de informação (dados ou controlo), que pode potencialmente recorrer a várias alternativas para

Atributos	Operações	Responsabilidades
Capacidade	Ir para piso	Transportar pessoas e
Piso actual	Parar	bens para o piso desejado.
Sentido de deslocamento	Abrir porta	
Ano de fabrico	Fechar porta	
Ano de instalação		

Tabela 3.1: Os atributos, as operações e as responsabilidades dum elevador.

implementação [Douglass, 1998, pág. 25]:

1. Chamada duma função.
2. Primitiva dum sistema operativo de tempo-real para envio de correio.
3. Interrupção.
4. *Rendez-vous* da linguagem Ada.
5. Chamada a um procedimento remoto (RPC) num sistema distribuído.

Durante a fase de análise, são identificadas as mensagens que podem transitar entre os objectos que compõem o sistema. É só nas fases de concepção e implementação que têm de se definir o tipo de sincronização e os requisitos temporais para cada mensagem.

3.2.6 Polimorfismo

O *polimorfismo* significa que a mesma operação pode apresentar comportamentos distintos para classes diferentes. Por exemplo, a operação roda comporta-se, seguramente, de modo distinto nas classes DiscoCompacto e Quadrado[9]. Se uma dada operação é polimórfica, então existe mais do que um método que a implementa. Em termos práticos, e segundo uma outra perspectiva, o polimorfismo significa também que o emissor duma dada mensagem não precisa de conhecer a classe a que pertence o objecto receptor.

Uma operação é vista como uma abstracção do comportamento idêntico que se verifica em diversos tipos de objectos. Cada objecto "conhece" como desempenhar as suas próprias operações. Numa linguagem orientada ao objecto, o método específico que implementa a operação é escolhido automaticamente, com base no nome da operação e na classe do objecto, pelo que o utilizador duma dada operação não necessita de se envolver explicitamente na escolha do método para implementar uma dada operação polimórfica. Este facto permite que novas classes de objectos possam ser acrescentadas, sem necessidade de alterar o código já existente, desde que sejam disponibilizados métodos para cada operação das novas classes. O polimorfismo conjuntamente com o encapsulamento (secção 3.2.4) podem ser considerados os principais responsáveis por uma das características mais significativas das linguagens orientadas ao objecto: a programação incremental ou diferencial [Thomas, 1989].

Outro conceito da orientação ao objecto, intrinsecamente relacionado com o polimorfismo, é a *sobreposição de operadores* (*operator overloading*) que se refere à possibilidade de se usar o mesmo símbolo para propósitos distintos, consoante o contexto em que se insere a sua utilização [Graham, 1991, pág. 14]. Por exemplo, a mensagem multiplicar(5) enviada a um número inteiro

[9] O uso do mesmo nome para operações semanticamente diferentes, mesmo que de classes distintas, deve evitar-se, uma vez que pode introduzir alguma confusão [Rumbaugh et al., 1991, pág. 25]. Por exemplo, o rodar dum disco compacto é conceptualmente diferente do rodar uma figura geométrica.

(ou, dito duma forma mais precisa, a uma instância da classe dos números inteiros) determina que seja invocado um método distinto àquele que o seria se a mesma mensagem fosse enviada a um número real. O uso do mesmo símbolo (no caso a cadeia de caracteres "soma") torna-se conveniente, pois facilita a compreensão e torna a linguagem mais simples de ler e aprender. Numa linguagem que não suporte polimorfismo, será obrigatório encontrar nomes diferentes para a mesma operação aplicada em tipos diferentes, como por exemplo multiplicarInt, multiplicarReal, multiplicarVector, multiplicarMatriz, para multiplicar um valor a um objecto do tipo inteiro, real, vector e matriz, respectivamente.

O polimorfismo representa um conceito da teoria dos tipos, no qual um dado nome (uma variável ou uma função) pode corresponder a objectos de várias classes relacionadas por uma superclasse comum. O polimorfismo é possível quando se conjugam os mecanismos de herança e *ligação dinâmica*[10] e é, conjuntamente com o conceito de abstracção, o elemento que distingue a programação orientada ao objecto da programação mais tradicional usando tipos abstractos de dados.

3.2.7 Herança e hierarquia

A *herança* é definida como o mecanismo que facilita a construção de novas classes (subclasses) a partir de outras classes (superclasses) e permite a especialização e a generalização de componentes. A herança é um mecanismo valioso e poderoso que facilita a reutilização e, ao mesmo tempo, permite expressar os aspectos comuns entre classes de objectos. À herança também se dá o nome de relação "é_do_tipo" ou "é_um" ("*is_a_kind_of*" ou "*is_a*").

A criação de variantes de classes, sem utilizar o mecanismo de herança disponível em modelação orientada ao objecto, pode ser conseguida por "cópia e modificação". Porém, daqui resultam inúmeras versões que diferem ligeiramente, contêm redundância e cuja manutenção se torna progressivamente dificultada, à medida que mais classes vão sendo acrescentadas. A coerência entre classes pode ser mantida, mas o "preço" a pagar é demasiado alto, visto que as modificações têm de realizar-se em muitos locais. Com uma *hierarquia* de classes e um mecanismo de herança garante-se uma maior reutilização das classes, devido a uma gestão mais facilitada [Nebel e Schumacher, 1996].

A herança inclui quatro aspectos principais [Coad e Yourdon, 1991, pág. 47–8]:

1. Uma subclasse herda automaticamente os atributos das suas superclasses.
2. Uma subclasse herda automaticamente as operações das suas superclasses.
3. Uma subclasse pode adicionar novos atributos àqueles existentes nas superclasses.
4. Uma subclasse, relativamente aos métodos definidos nas superclasses, pode reescrevê-los ou adicionar novos.

Estes dois últimos aspectos são os que permitem que uma subclasse se possa diferenciar da sua superclasse. Considerando, por exemplo, a existência duma classe Registo, que disponibiliza os métodos ler e escrever, pode, com o mecanismo de herança, criar-se, muito rapidamente, uma subclasse RegReset. Para tal, há apenas que adicionar um novo método iniciar, aproveitando todos os atributos e métodos definidos na superclasse. Pode igualmente criar-se, a partir da classe Registo, uma subclasse RegPilha que disponibiliza duas novas operações (incrementar e

[10] A ligação dinâmica (*dynamic* ou *late binding*) indica a possibilidade de determinar dinamicamente, em tempo de execução, a classe dum objecto. Em oposição, o termo *ligação estática* (*static* ou *early binding*) indica que a classe dum objecto é calculada, em tempo de compilação.

decrementar), para permitir alterar o conteúdo do atributo valor. A fig. 3.5 ilustra este exemplo, usando novamente a notação UML.

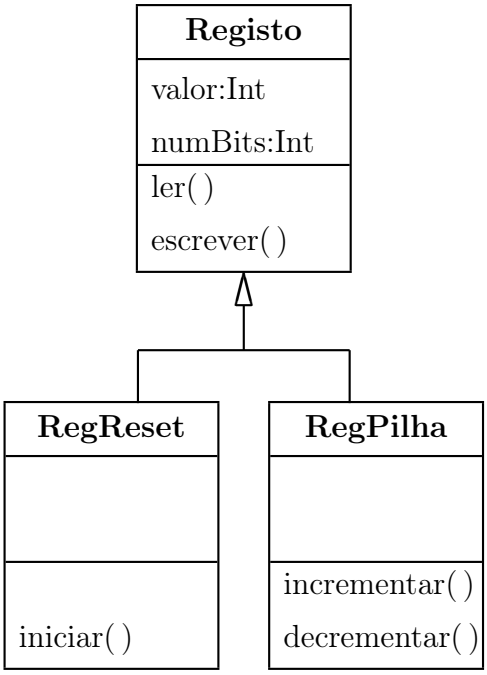


Figura 3.5: Exemplo do mecanismo de herança.

Considerou-se, anteriormente, o uso de *herança não estrita* , que permite, a uma subclasse, alterar e apagar propriedades da sua superclasse. Em contraponto, a *herança estrita* força a que as propriedades duma classe sejam todas retidas pela subclasse, permitindo unicamente a adição de novas propriedades.

O mecanismo que permite redefinir o comportamento dum método herdado chama-se *reescrita de métodos*. Desta forma, o método reescrito tem o mesmo nome do herdado, mas comporta-se diferentemente, conforme a classe do objecto em causa. Este mecanismo de reescrita implementa um comportamento polimórfico.

À medida que a hierarquia de classes vai sendo enriquecida, nomeadamente com a introdução de mais classes, a estrutura e o comportamento idênticos de várias classes tendem a ser definidos numa superclasse comum. Por esta razão, se diz que a herança é uma hierarquia de generalizações/especializações. As superclasses representam generalizações de abstracções, ao passo que as subclasses representam especializações, nas quais os atributos e os métodos das superclasses podem ser adicionados, modificados ou apagados.

Os termos herança, generalização e especialização referem-se todos a aspectos relacionados com o mesmo conceito. A *generalização* refere-se à relação entre classes, enquanto que a herança se refere ao mecanismo que permite a partilha de atributos e operações, usando a relação de generalização. Generalização e especialização são dois pontos de vista diferentes da mesma relação. A generalização deriva do facto da superclasse generalizar as características das subclasses e a especialização refere-se ao facto das subclasses refinarem (ou especializarem) a superclasse.

Em 1986, Booch reconhecia que a herança era um mecanismo importante, mas não a classificava como indispensável para que algo fosse considerado orientado ao objecto, ou seja, na sua opinião, uma linguagem de programação, por exemplo, sem o mecanismo de herança podia

considerar-se orientada ao objecto, desde que incluísse toda as outras características consideradas fundamentais [Booch, 1986]. Booch viria a reconsiderar a sua opinião, 5 anos mais tarde, ao afirmar taxativamente que "*Programming without inheritance is distinctly not object-oriented*" [Booch, 1991, pág. 36].

Alguns dos benefícios que advêm duma utilização apropriada do mecanismo de herança são os seguintes [Budd, 1997, pág. 143–5]:

- **Reutilização**: Quando o comportamento duma dada classe é herdado duma outra, o código que descreve esse comportamento não precisa de ser escrito.
- **Partilha de código**: Pode ocorrer a vários níveis. Por um lado, vários utilizadores podem, em projectos distintos, usar as mesmas classes (vistas como componentes). Por outro lado, a partilha pode dar-se quando duas ou mais classes desenvolvidas por um programador herdam da mesma superclasse.
- **Consistência dos comportamentos e das interfaces**: Quando várias classes herdam da mesma superclasse, há a garantia, a menos que haja reescrita de métodos, que o comportamento vai ser exactamente o mesmo em todas elas. Este facto permite igualmente assegurar que objectos semelhantes tenham a mesma interface.
- **Componentes de software**: A herança facilita a utilização e a construção de componentes de software reutilizáveis.
- **Prototipagem rápida**: Quando uma importante percentagem do sistema é construída com componentes reutilizáveis, o esforço de desenvolvimento pode ser dirigido nas partes do sistema que são novas, o que possibilita gerar sistemas duma forma mais rápida e fácil.
- **Ocultação de informação**: Quando o programador reutiliza um componente, precisa somente de conhecer a sua funcionalidade visível exteriormente e a sua interface, sendo dispensável o conhecimento dos algoritmos usados para obter a implementação do componente.

No entanto, como seria de esperar, o uso de herança também acarreta algumas penalizações que a seguir se indicam [Budd, 1997, pág. 145–6]:

- **Velocidade de execução**: Os métodos herdados, pelo facto de manipularem objectos de várias classes, são, geralmente mais lentos que o código específico que se poderia ter escrito para uma dada classe.
- **Tamanho dos programas**: O uso de bibliotecas de componentes determina, normalmente, que o sistema tenha um tamanho superior, se comparado com sistemas construídos de raiz. Este facto pode ser importante se houver limites quanto à memória disponível, o que acontece muitas vezes com alguns microprocessadores usados no projecto de sistemas embebidos.
- **Passagem de mensagens**: A passagem de mensagens é um mecanismo mais pesado do que a invocação duma função, o que implica novamente um tempo de execução superior.
- **Complexidade dos programas**: Apesar da orientação ao objecto permitir atacar a complexidade dos sistemas, o abuso da herança pode implicar a substituição duma forma de complexidade por outra. De facto, captar o fluxo de controlo dum programa que usa herança pode requerer várias travessias, para cima e para baixo, na hierarquia de classes, dificultando a legibilidade do programa. Esta problema é conhecido como o *problema ió-ió* [Taenzer, 1989].

O conceito de *herança múltipla* consiste na possibilidade duma subclasse ter mais do que uma superclasse, no nível imediatamente superior. O uso de herança múltipla mostra-se vantajoso em determinadas situações. Considere-se, por exemplo, duas classes RegistoEsq e RegistoDir,

que definem registos de deslocamento (*shift registers*) para a esquerda e para a direita, respectivamente. Esta duas classes podem ser usadas como superclasses, permitindo obter uma classe RegistoBi que suporta ambos os sentidos de deslocamento, sem a necessidade de escrever nenhum método novo, como a fig. 3.6 pretende ilustrar. Há um amplo reconhecimento que a herança múltipla pode, em determinadas situações, revelar-se útil, mas trata-se, porém, dum conceito bastante difícil de implementar e, conceptualmente, complicado de manipular [Stroustrup, 1987] [Szyperski, 1998, pág. 98].

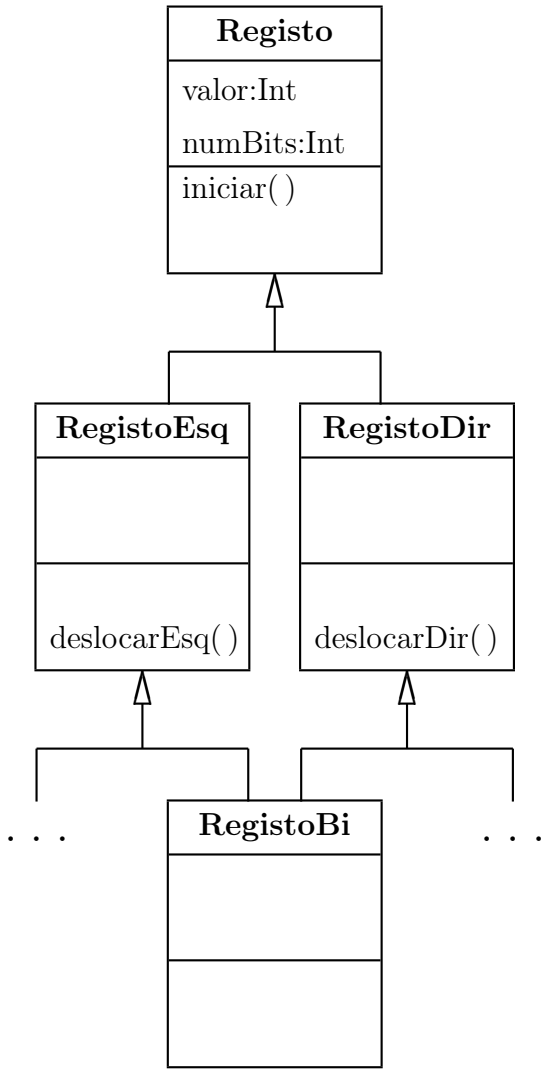


Figura 3.6: Exemplo do mecanismo de herança múltipla.

A herança múltipla reduz o entendimento da hierarquia de classes. Este problema torna-se ainda mais evidente, quando as superclasses têm duas operações com o mesmo nome. Neste caso, é preciso dar um novo nome a uma das operações ou escolher qual das duas é relevante para a subclasse.

Rumbaugh *et al.* apresentam algumas técnicas para reestruturar uma hierarquia de classes contendo herança múltipla numa outra equivalente, mas apenas usando o mecanismo de herança simples [Rumbaugh et al., 1991, pág. 67–9]. Estas técnicas revelam-se essenciais no caso de se pretender implementar uma hierarquia de classes que usa o mecanismo de herança múltipla, recorrendo a linguagens ou utilitários que disponibilizam somente o mecanismo de herança

simples.

O exemplo mostrado na fig. 3.6 permite evidenciar o carácter recursivo do mecanismo de herança. De facto, uma dada classe herda de todas as suas superclasses (de todos os seus antecessores) e não apenas das superclasses do nível imediatamente superior. Assim, a classe RegistoBi, além de herdar das suas classes imediatamente superiores (RegistoEsq e RegistoDir), também herda das classes imediatamente acima destas (Registo) e assim sucessivamente até se chegar à raiz da hierarquia. A fig. 3.7 mostra, sem recorrer ao mecanismo de herança, para cada uma das quatro classes, quais são os seus atributos e as suas operações. As fig. 3.6 e 3.7 podem representar as perspectivas distintas que têm, em relação à estrutura de classes, respectivamente, quem a implementou e quem a usa.

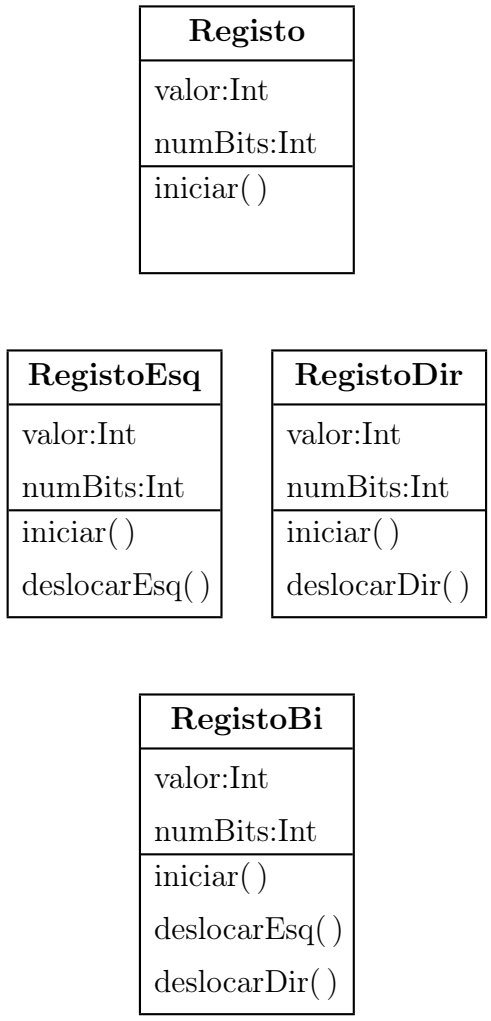


Figura 3.7: Reformulação da figura anterior, sem recurso ao mecanismo de herança.

Encapsulamento e herança são conceitos essenciais em sistemas descritos segundo a abordagem orientada ao objecto, embora sejam, de alguma forma, incompatíveis entre si. O encapsulamento significa que o cliente duma classe não deve ver a representação interna desta. Por outro lado, se se entender uma subclasse como cliente da sua superclasse (na medida em que usa alguns dos seus atributos e operações), então o cliente tem acesso completo à representação interna da superclasse. Esta contradição resulta do facto de existirem 3 tipos distintos de clientes: aqueles que usam a classe através da sua interface, aqueles que usam a classe através

do mecanismo de herança e aqueles que implementam a classe [Jacobson et al., 1992, pág. 99].

3.2.8 Agregação ou composição

As relações inter-objectos que se expressam, durante a utilização do sistema, através da troca de mensagens entre eles, designam-se por *associações*. Assim, quando um objecto recorre aos serviços dum outro, deve estabelecer-se entre eles uma associação.

A *agregação* (também designada por *composição*) representa uma forma especial de associação, que se verifica quando um objecto contém, física ou logicamente, outros. O objecto incluído é designado por *componente* ou *parte* e o objecto mais amplo (aquele que é constituído pelos componentes) diz-se *agregado* ou *composto*. A agregação e a composição também são designadas por relações "é_parte_de" ("*is_part_of*").

Um agregado é mais do que a simples soma das suas partes. Um agregado tem os seus próprios atributos e operações que não têm de estar directamente relacionados com as partes. Muitas vezes, é conceptualmente conveniente, para permitir reduzir a complexidade, agrupar um conjunto de objectos num agregado, a que se atribuem novas propriedades, passando a ver-se o agregado como um novo objecto. Assim, considera-se que um *agregado* é uma colecção de partes que juntas formam algo novo que é mais do que a soma dessas partes. Um agregado introduz novas características que pertencem ao todo e não às partes e constitui uma abstracção que permite ignorar (ou esconder) certos pormenores.

Como exemplo dum agregado considere-se um computador que é composto por monitor, chassis, rato e teclado. O computador tem atributos próprios que não dependem dos seus componentes: nome lógico, endereço IP, local onde está instalado, dono, preço, etc. Este exemplo também permite mostrar que a agregação é um mecanismo que pode aplicar-se recursivamente, uma vez que o chassis é composto pela placa-mãe, unidade de disquetes, disco duro, etc.

Apesar de ser um conceito extremamente intuitivo, a agregação nem sempre é bem utilizada na modelação de sistemas. O grande problema resulta da dificuldade em distinguir entre associação e agregação, que pode comprovar-se pelo facto de alguns dos especialistas mais conceituados na área de modelação orientada ao objecto terem visões diametralmente opostas e, em certo sentido, contraditórias. Coad e Yourdon dão, como exemplo de agregação, a relação entre uma organização e os seus empregados [Coad e Yourdon, 1991, pág. 92–3], ao passo que Rumbaugh *et al.* afirmam exactamente o contrário: "*A company is not an aggregation of its employees*" [Rumbaugh et al., 1991, pág. 58]. Assim, parece natural concluir que não há uma opinião consensual sobre quais as diferenças substantivas entre associação e agregação.

A notação UML, que será apresentada na secção 3.3, faz uma distinção entre agregação e composição, que será adoptada neste trabalho. As partes dum agregado podem ser partilhadas por outros agregados, o que não sucede com um composto. Exemplificando, se um clube for composto pelos seus sócios, é possível uma mesma pessoa ser sócia de vários clubes (trata-se duma agregação); se uma pessoa for composta pela cabeça, pelo tronco e pelos membros, não deve ser possível partilhar esses "componentes" por várias pessoas (trata-se duma composição). Por outro lado, a criação e destruição dos componentes duma composto é da responsabilidade deste (i.e. a existência dos componentes só tem significado se o composto existir), o que não se verifica num agregado, quanto mais não seja devido à possibilidade de as suas componentes poderem ser partilhadas por outros agregados.

3.3 A notação UML

Ao longo da secção 3.2 já foram apresentados alguns exemplos concretos da notação utilizada para representar os diversos aspectos dos sistemas, a modelar segundo os princípios da decomposição por objectos. A notação seleccionada foi a versão 1.1 de UML (Unified Modeling Language) [Unified Modeling Language, 1997c] [Unified Modeling Language, 1997b] [Booch et al., 1999] [Rumbaugh et al., 1999], por ser a notação oficial da OMG (Object Modeling Group[11]) e por ser, parcialmente, suportada pela ferramenta OBLOG. UML é uma linguagem para expressar a funcionalidade, a estrutura e as relações de sistemas complexos. Nesta secção, não se pretende dar uma cobertura exaustiva da notação adoptada, tendo havido contudo o cuidado de documentar os seus aspectos mais relevantes.

A definição de UML inclui o respectivo meta-modelo [Unified Modeling Language, 1997a], ele próprio especificado em UML, o que elimina, caso se use uma linguagem formal para o definir, quaisquer ambiguidades e permite conhecer a semântica dos mecanismos de modelação disponibilizados pela linguagem. No entanto, o meta-modelo UML não está ainda totalmente definido de forma formal, pois a sintaxe dos construtores está especificada numa linguagem precisa, mas a respectiva semântica ainda só foi descrita em inglês corrente. Daí que, UML se possa classificar, actualmente, como semi-formal, uma vez que a estrutura da linguagem é rigorosa mas a sua semântica é ainda bastante informal [Övergaard e Palmkvist, 1998]. Contudo, as diversas propostas que já foram sugeridas no sentido de formalizar a semântica de UML, permitem antever que, num futuro não muito distante, UML terá associada uma base formal [Evans et al., 1998] [van Emde Boas, 1998] [Geisler et al., 1998] [Bacławski et al., 1998].

A notação UML é composta por um conjunto de diagramas que permitem descrever os aspectos mais relevantes dos sistemas a implementar segundo a abordagem orientada ao objecto. Partindo do princípio que os sistemas a desenvolver são complexos (caso não o fossem, não seria necessária uma metodologia de desenvolvimento, pois qualquer abordagem *ad-hoc* seria mais do que suficiente), cada um dos diagramas foca uma dada vista do sistema e propositadamente enfatiza alguns aspectos e negligencia outros.

Note-se que as notações gráficas, apesar de se pretenderem independentes de qualquer linguagem de programação, são, regra geral, fortemente influenciadas por aquela que os proponentes da respectiva metodologia adoptam para a fase de implementação. Este facto deve ser entendido como natural e positivo, uma vez que as notações gráficas não são mais do que uma camada com características pictóricas, a fim de facilitar o diálogo não só entre os clientes e os projectista, mas também entre os elementos que compõem a equipa de projecto. No entanto, quando a notação apresenta um conjunto básico de propriedades e mecanismos comuns à maioria das linguagens, o uso dessa notação pode, facilmente, generalizar-se.

Por exemplo, a notação apresentada em [Booch, 1991], apesar de poder considerar-se genérica, foi definida "à medida" para a linguagem ADA, tendo em conta as suas características, mas adapta-se igualmente bem a outras linguagens, como C++, OBJECTPASCAL, CLOS ou SMALLTALK. A notação UML é independente da linguagem de programação e do processo de desenvolvimento adoptados e, além disso, está dotada dum mecanismo de extensão (que usa *estereótipos*) que facilita a sua adaptação a situações que apresentem mecanismos não previstos inicialmente na notação. Um estereótipo é a classe duma entidade no meta-modelo UML.

Assim, no contexto deste trabalho, os seguintes diagramas UML foram considerados como

[11] Informações sobre esta organização estão disponíveis no seguinte URL: `www.omg.org`.

indispensáveis para especificar e documentar os vários aspectos que importa considerar na modelação dum sistema complexo:

- Diagramas de casos de uso (*use case diagrams*).
- Diagramas de classes (*class diagrams*).
- Diagramas de objectos (*object diagrams*).
- Diagramas de interacção (*interaction diagrams*).
- Diagramas de estados (*state diagrams*).

O quadro 3.2 pretende sumariar o propósito de cada um desses 5 diagramas (i.e. a perspectiva do sistema que cada um deles evidencia), no contexto do desenvolvimento de sistemas orientados ao objecto.

Diagrama	**Propósitos**
Diagrama de casos de uso	Mostrar um conjunto de casos de uso e de actores e as respectivas relações. Estes diagramas são importantes para organizar e modelar as funcionalidades do sistema.
Diagrama de classes	Apresentar um conjunto de conceitos, tipos e classes e respectivas relações.
Diagrama de objectos	Exibir um conjunto de instâncias e a forma como elas se relacionam. Como sucede com os diagramas de classes, a estrutura estática do sistemas é mostrada, mas a ênfase é colocada em configurações estáticas num dado instante.
Diagrama de interacção	Revelar como vários objectos colaboram (trocam mensagens) num dado caso de uso.
Diagrama de estados	Especificar o comportamento dum objecto, englobando muitos casos de uso.

Tabela 3.2: Diagramas UML usados no contexto do desenvolvimento de sistemas orientados ao objecto e respectivos propósitos.

De seguida, faz-se uma descrição breve dos 5 digramas atrás mencionados, tentando ilustrar os conceitos de modelação mais relevantes que cada um deles abarca.

3.3.1 Diagramas de casos de uso

Um *diagrama de casos de uso*, de que se mostra um exemplo para o elevador na fig. 3.8, utiliza, como elementos básicos, casos de uso (representados por elipses) e actores (representados por bonecos), que indicam, respectivamente, quais as funcionalidades que esse sistema deve desempenhar e o que existe fora dele. Como outro exemplo de casos de uso, tome-se a interacção dum utilizador com o seu processador de texto; assim, dois casos de uso serão os seguintes: "sublinhar uma porção de texto" e "verificar a correcção ortográfica do texto". Repare-se no uso de verbos para caracterizar os casos de uso, o que indica que, associada a estes, há uma dada operacionalidade (funcionalidade).

Um *caso de uso* é uma típica interacção entre um utilizador e um sistema computacional (ou apenas sistema, num sentido mais lato) [Jacobson et al., 1992, pág. 159]. Os casos de uso são

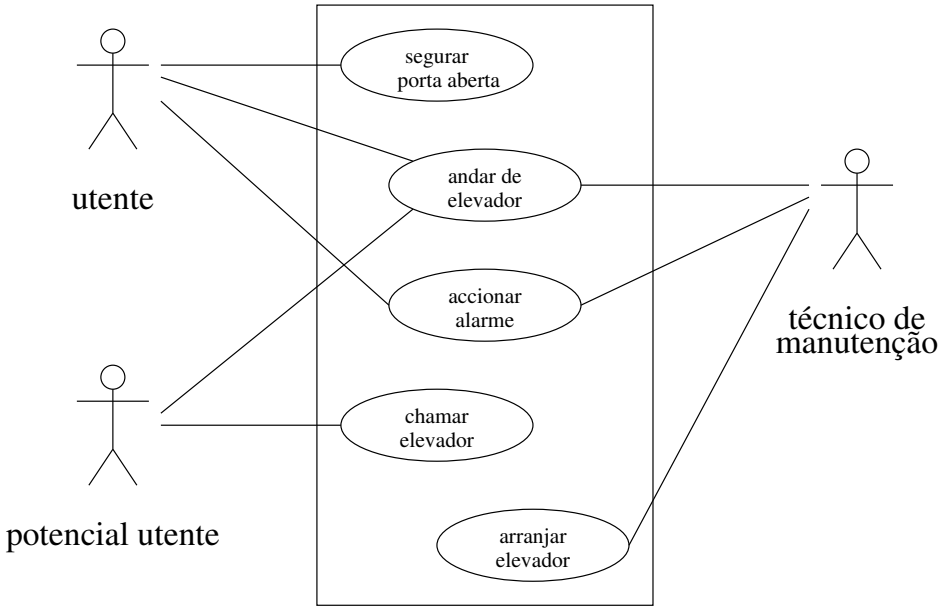


Figura 3.8: Um diagrama de casos de uso.

considerados uma ferramenta muito importante, no contexto do desenvolvimento orientado ao objecto, porque permitem captar, junto dos clientes, os requisitos do utilizador, dado que o vocabulário usado é o dos clientes e não o dos fabricantes. Os casos de uso dum sistema constituem uma decomposição funcional do comportamento desse sistema, sem lhe impor qualquer estrutura interna. UML dá uma grande importância aos diagramas de casos de uso, uma vez que é com base neles que, segundo os seus proponentes, se pode planear e sustentar todo o desenvolvimento do sistema em causa.

Um *actor* representa um papel que um dado utilizador pode ter em relação a um determinado sistema, quando com ele interactua. Os casos de uso são realizados por actores, podendo um mesmo actor desempenhar vários casos de uso e sendo também possível que um dado caso de uso seja executado por vários actores. Note-se que várias pessoas podem ser representadas pelo mesmo actor no caso de uso, como se pode constatar pelo facto do elevador ter sido construído para servir vários clientes e não apenas um. Um actor é assim uma representação abstracta dum tipo de pessoa que interage com o sistema. Note-se também que a mesma pessoa pode desempenhar vários papéis relativamente ao mesmo sistema, papéis esses que darão origem a actores distintos no respectivo diagrama. No exemplo do elevador, o técnico de manutenção pode também ele, em algumas situações, ser utilizador e, num exemplo mais complexo, uma mesma pessoa pode ser, relativamente a uma dada empresa, administrador, accionista e cliente. O símbolo dos actores tem o aspecto duma figura humana muito estilizada, mas isso não implica que os actores tenham necessariamente que ser humanos, podendo também representar outros sistemas (entidades externas) que interactuam com o sistema em causa (daqui, mais uma vez, resulta a definição de sistema embebido sugerida na secção 1.1).

A identificação dos actores do sistema facilita a definição das funcionalidades do sistema, feita através da especificação dos casos de uso. Um caso de uso consiste numa forma particular de usar o sistema, representando parte da sua funcionalidade total. Cada caso de uso constitui um conjunto completo de eventos, iniciado por um actor, e explicita a interacção que se pode

observar entre esse actor, o sistema e, eventualmente, outros actores. O conjunto de todos os casos de uso permite especificar todas as formas distintas de interagir com o sistema.

Ao definirem-se os actores e os casos de uso que compõem o sistema, está a delimitar-se o sistema, i.e. a definir o âmbito do sistema que se pretende desenvolver, o que se revela bastante importante no início do processo de desenvolvimento. Muitas metodologias (estruturadas, sobretudo) sugerem ou impõem mesmo, como uma das primeiras tarefas da análise, a criação dum *diagrama de contexto*, cuja utilidade primeira é demarcar claramente a fronteira do sistema e mostrar quais as entidades que com ele interagem. Um diagrama de contexto define inequivocamente o ambiente que rodeia o sistema em desenvolvimento e, ao usarem-se casos de uso, a tarefa da criação do diagrama de contexto está, em forte medida, a ser realizada.

UML define dois tipos possíveis de relação entre os casos de uso: ≪extends≫ e ≪uses≫ [Övergaard, 1998] [Övergaard e Palmkvist, 1998]. A relação ≪extends≫ indica que o caso de uso de onde parte a seta estende (i.e. especializa) a funcionalidade do caso de uso destino, ou seja, o primeiro é similar ao segundo, mas acrescenta funcionalidade. O exemplo da fig. 3.9(a) mostra que "Editar dados confidenciais" é uma forma especializada de "Editar dados". Este caso de uso descreve a situação de edição normal de informação, em que são indicados vários dados de carácter genérico (por exemplo, o nome, a morada e o tipo de sangue do paciente), enquanto que o primeiro se refere a uma situação em que, além dessas informações, são introduzidos outros dados de carácter mais confidencial (por exemplo, estado clínico ou a indicação duma doença). A relação ≪uses≫ indica que o caso de uso origem depende da funcionalidade disponibilizada pelo caso de uso destino, mas que não há necessariamente similitude entre os dois casos de uso. O exemplo da fig. 3.9(b) mostra que para ser possível mostrar o nível de poluição do ar é obrigatório, primeiramente, captar o nível de poluição do ar.

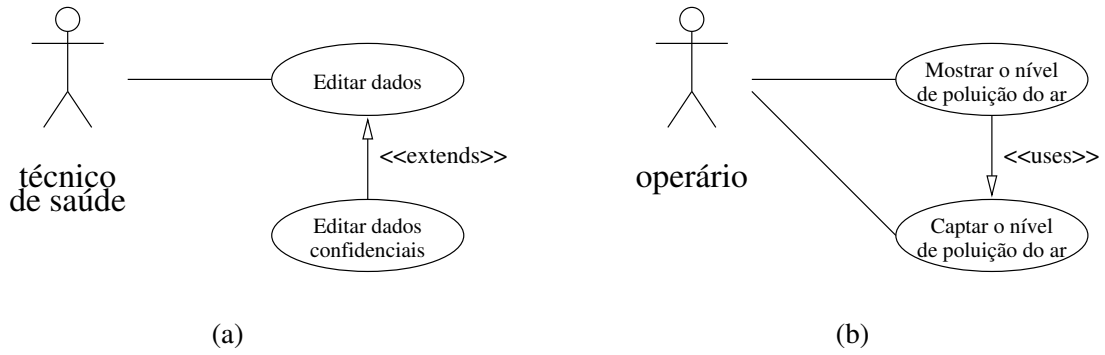


Figura 3.9: As relações entre casos de uso: (a) ≪extends≫ e (b) ≪uses≫.

Estas duas relações implicam situações distintas no que respeita à ligação com os actores [Fowler e Scott, 1997, pág. 50]. Relativamente a ≪extends≫, os actores têm relação com o caso de uso base (aquele que está a ser estendido) e assume-se que também têm ligação a todas as extensões. A fig. 3.9(a) reflecte este facto, pois o caso de uso "Editar dados confidenciais" por estender "Editar dados", não carece de ligação ao técnico de saúde, dado ser implícita essa ligação. Quanto à relação ≪uses≫, habitualmente não existem actores associados com o caso de uso comum, mas nas situações em que há, não se considera implicitamente que haja ligação com os outros casos de uso. A fig. 3.9(b) ilustra esta hipótese, já que o caso de uso "Captar o nível de poluição do ar" necessita de uma ligação ao actor operário, para indicar que ele pode executar esta funcionalidade.

Um termo muito ligado aos casos de uso é cenário, que, por vezes, é usado como sinónimo. Na terminologia UML, um *cenário* refere-se apenas a um dado caminho dentro do caso de uso, determinado por uma combinação particular de condições, ou seja, um cenário é uma instância dum caso de uso, da mesma forma que um objecto representa uma instância duma determinada classe. Significa isto que, para especificar um caso de uso, é necessário considerar vários cenários. Como a fig. 3.8 ilustra, para o elevador, existirá um caso de uso para representar a chamada do elevador por parte dum potencial passageiro ("chamar elevador"), podendo este caso de uso ter associados vários cenários, de que se indicam alguns exemplos a seguir:

- O elevador está disponível no mesmo piso do passageiro.
- O elevador está disponível, mas num piso diferente daquele onde está o passageiro.
- O elevador está a subir (descer) e vai passar no piso onde está o passageiro que, no entanto, pretende descer (subir).
- O elevador está a descer (subir) e vai passar no piso onde está o passageiro que pretende igualmente descer (subir).
- O elevador está a movimentar-se mas não vai passar no piso onde está o passageiro.
- O alarme foi accionado por outro passageiro que se encontra dentro do elevador.

Os cenários são modelados, em UML, através de diagramas de sequência ou diagramas de colaboração (ver diagramas de interacção mais à frente).

3.3.2 Diagramas de classes

Para sistemas orientados ao objecto, são necessários *diagramas de classes*, para indicar as classes existentes e as suas relações. Este tipo de diagrama é, em alguma das suas variantes, sempre contemplado por todas as metodologias de desenvolvimento orientado ao objecto, uma vez que o conceito de classe é um elemento fundamental para os sistemas desenvolvidos segundo este paradigma.

Um diagrama de classes tem por objectivo evidenciar a estrutura estática de conceitos, tipos e classes. Os conceitos mostram como os utilizadores vêem o domínio da aplicação, independentemente da forma como eles são, na prática, implementados. Estes diagramas, de que se mostra um exemplo na fig. 3.10, permitem também descrever os tipos de objectos (as classes) que o sistema pode contemplar e as formas de inter-relação entre eles. Podem ainda ser mostrados os atributos e as operações de cada uma das classes, caso tal seja relevante para o nível de abstracção em causa, como foi considerado no exemplo da fig. 3.6.

A mensagem é a unidade fundamental de comunicação entre objectos. Se dois objectos comunicam entre si, há entre eles uma ligação ou associação. Em UML, existem basicamente 4 tipos de relações entre objectos, que podem ser mostradas entre as classes nos respectivos diagramas:

- **Associação**: Representa uma relação entre objectos que se manifesta em tempo de execução através da troca de mensagens entre eles (por exemplo, um aluno estuda numa universidade ou uma companhia emprega vários trabalhadores). Quando um objecto usa os serviços dum outro, devem ligar-se estes com uma associação. As associações são representadas, em UML, por linhas e, a menos que se explicite o sentido da comunicação através dum triângulo, consideram-se bidireccionais.
- **Agregação**: Representa uma forma especial de associação, que se verifica quando um objecto contém, física ou logicamente, outros. À classe incluída dá-se o nome de *componente* ou *parte* e a classe mais ampla (aquela que inclui os componentes) diz-se *agregado* ou *composto*. Em UML, a agregação representa-se por uma linha com um losango junto

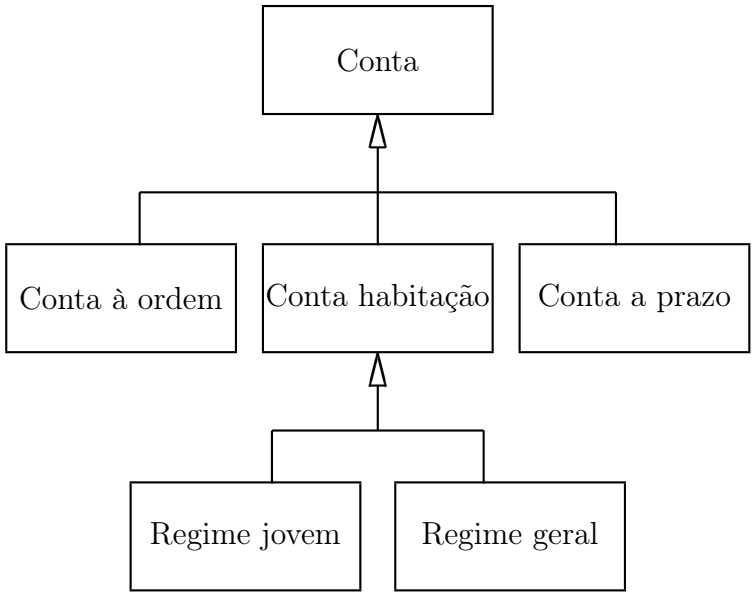


Figura 3.10: Um diagrama de classes.

do agregado. É possível que os componentes possam ser partilhados por vários agregados, caso assim se ache conveniente.

- **Composição**: É uma forma mais restrita de agregação, em que os componentes são mostrados por inclusão gráfica no agregado; alternativamente, pode ser usado um losango cheio, mas a primeira representação é preferível. Os componentes dum agregado por composição não podem ser partilhados por outros agregados. O agregado é responsável pela criação e destruição dos seus componentes.
- **Generalização**: Verifica-se quando uma classe é uma especialização duma outra. A subclasse herda todas as características da superclasse, podendo adicionar novos atributos ou operações. Em UML, esta relação entre classes é representada por uma seta que parte da subclasse e termina na superclasse.

A fig. 3.11 mostra as relações entre classes anteriormente descritas. Cada relação tem dois *papéis*[12], representando cada um deles a forma como cada objecto participa na relação e fornecendo um sentido de leitura possível da relação. A inclusão dos papéis é opcional, mas é recomendada quando clarifica a relação entre os objectos. O uso dos papéis torna-se necessário para relações entre objectos da mesma classe ou quando dois objectos têm mais do que uma relação. Regra geral, não é necessário identificar os dois papéis da mesma relação, pois um é o conjugado do outro. Por exemplo, se um papel é "controla" o outro será "é controlado por"; neste caso, o primeiro papel está escrito na voz activa e o segundo na voz passiva.

Os números ou símbolos no fim das relações, a que se dá o nome de *multiplicidade*, indicam o número de objectos da respectiva classe que participam na relação. A multiplicidade indica os limites inferior e superior para o número de objectos que participam na relação.

[12] *Role* é o termo usado na terminologia UML.

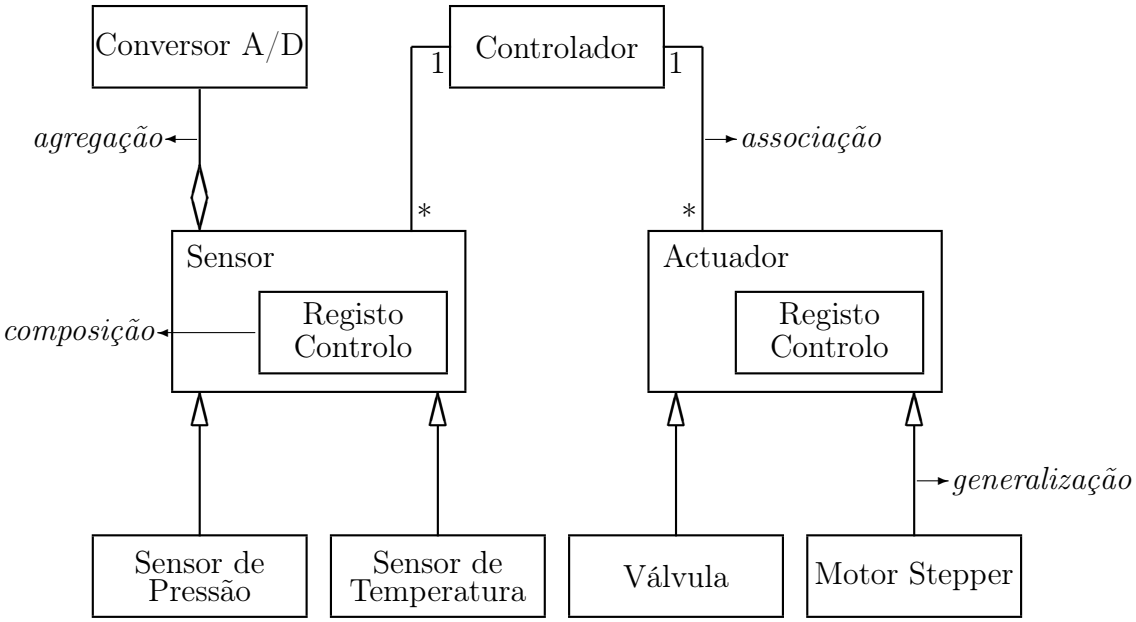


Figura 3.11: As várias relações possíveis entre classes, segundo a notação UML [Douglass, 1998, pág. 131].

3.3.3 Diagramas de objectos

Para documentar os objectos concretos que compõem o sistema, bem como as suas inter-relações, são usados *diagramas de objectos*. Estes são, por natureza, estáticos, no sentido em que apenas mostram um conjunto de objectos que trocam mensagens entre eles, omitindo o fluxo de controlo e a ordem dos eventos.

Os diagramas de objectos são idênticos aos diagramas de classes, exceptuando o facto de mostrarem instâncias (ou objectos) em vez de classes. Os objectos são desenhados como rectângulos, sendo os nomes sublinhados para mais facilmente os distinguir de classes. A fig. 3.12 apresenta um exemplo dum diagrama de objectos, onde se pode constatar a semelhança com os diagramas de classe.

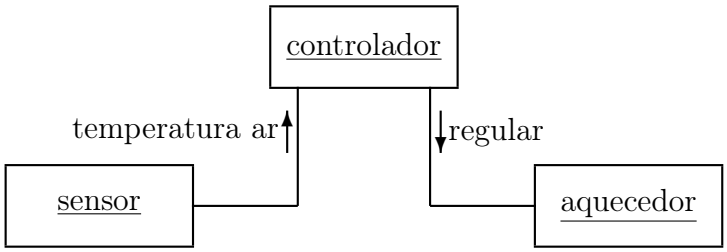


Figura 3.12: Um diagrama de objectos.

3.3.4 Diagramas de interacção

São também necessárias representações gráficas que permitam captar os aspectos dinâmicos relativos à troca de mensagens entre objectos, a que se chamam *diagramas de interacção*.

Como se verá mais à frente, os diagramas de estado associados a algumas classes não servem para este efeito, pois apenas descrevem as mudanças internas de estado das instâncias duma determinada classe e não entre um conjunto de objectos (normalmente de classes diferentes).

Um diagrama de interacção representa uma instância dum caso de uso [Booch, 1996, pág. 112]. Os diagramas de interacção descrevem a forma como um grupo de objectos comunica entre si. Tipicamente, um diagrama de interacção capta o comportamento dum dado cenário, mostrando os objectos e as mensagens que são trocadas entre eles, nesse caso de uso [Douglass, 1998, pág. 76]. Existem, em UML, dois tipos diferentes de diagramas de interacção: diagramas de sequência e diagramas de colaboração.

Estes diagramas permitem captar os requisitos do sistema, podendo portanto ser utilizados na fase de análise. Podem também usar-se, durante o teste do sistema, para comparar o funcionamento real do sistema (ou do protótipo, ou do modelo executável) com aquele que foi especificado.

Diagramas de sequência

Um *diagrama de sequência* mostra a sequência de mensagens trocadas entre objectos. A fig. 3.13 apresenta um exemplo dum diagrama de sequência e contém os elementos principais que podem ser encontrados neste tipo de diagrama.

Uma linha vertical representa um objecto, sendo o respectivo nome indicado em cima ou em baixo. As setas horizontais são as mensagens. Cada mensagem parte dum objecto (o responsável pela sua criação) e termina num outro (aquele a quem a mensagem se dirige). A indicação do nome da mensagem faz-se por cima da seta.

A leitura temporal do diagrama faz-se de cima para baixo, o que significa que, na fig. 3.13, a mensagem "Pedido para subir" é enviada antes da mensagem "Pedido para descer". É importante notar que o eixo temporal não tem associada qualquer escala, evidenciando apenas a relação de ordem (antes ou depois) entre os acontecimentos.

Do lado esquerdo do diagrama podem incluir-se algumas anotações de texto, com o propósito, entre outros, de identificar condições iniciais, acções e actividades não evidentes pela leitura do resto do diagrama.

A maioria dos diagramas de sequência contém apenas os elementos indicados anteriormente: objectos, mensagens, períodos de inactividade (implícitos) e anotações textuais. Os diagramas de sequência contêm mais quatro mecanismos que se mostram úteis na modelação de alguns sistemas: identificadores de eventos, marcas temporais, marcas de estado e mensagens difundidas. Estes mecanismos serão explicados recorrendo ao exemplo que a fig. 3.14 ilustra.

Junto a cada mensagem, pode incluir-se um *identificador de evento* que indica qual o evento responsável pelo envio da referida mensagem. Por exemplo, a figura mostra que o envio da mensagem Mens1 se deve à ocorrência do evento ev1.

Para sistemas de tempo-real, em que há restrições temporais a cumprir, torna-se imprescindível a utilização de *marcas temporais*. Existem duas formas distintas de indicar as marcas temporais. A primeira forma consiste em usar uma barra com um tempo limite, definindo assim o tempo que medeia entre os eventos que se encontram nos extremos da barra. Na figura, esta forma é usada para definir que o tempo entre os eventos ev5 e ev6 é 2s. A segunda forma consiste na indicação, entre chavetas, de expressões relacionais, para explicitar restrições temporais. Esta forma é usada, na figura, para definir que o tempo entre os eventos ev2 e ev1 não pode exceder

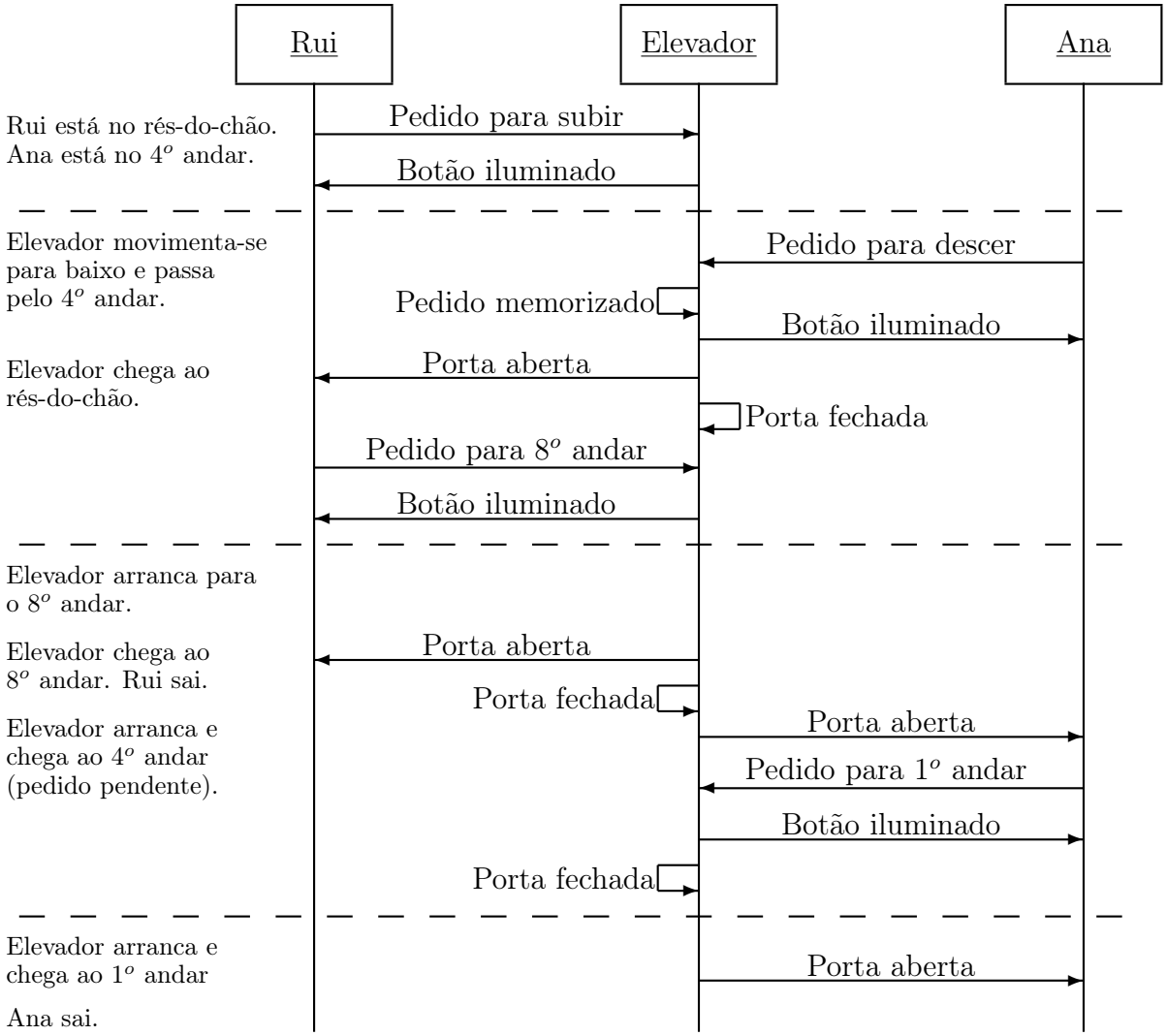


Figura 3.13: Um diagrama de sequência.

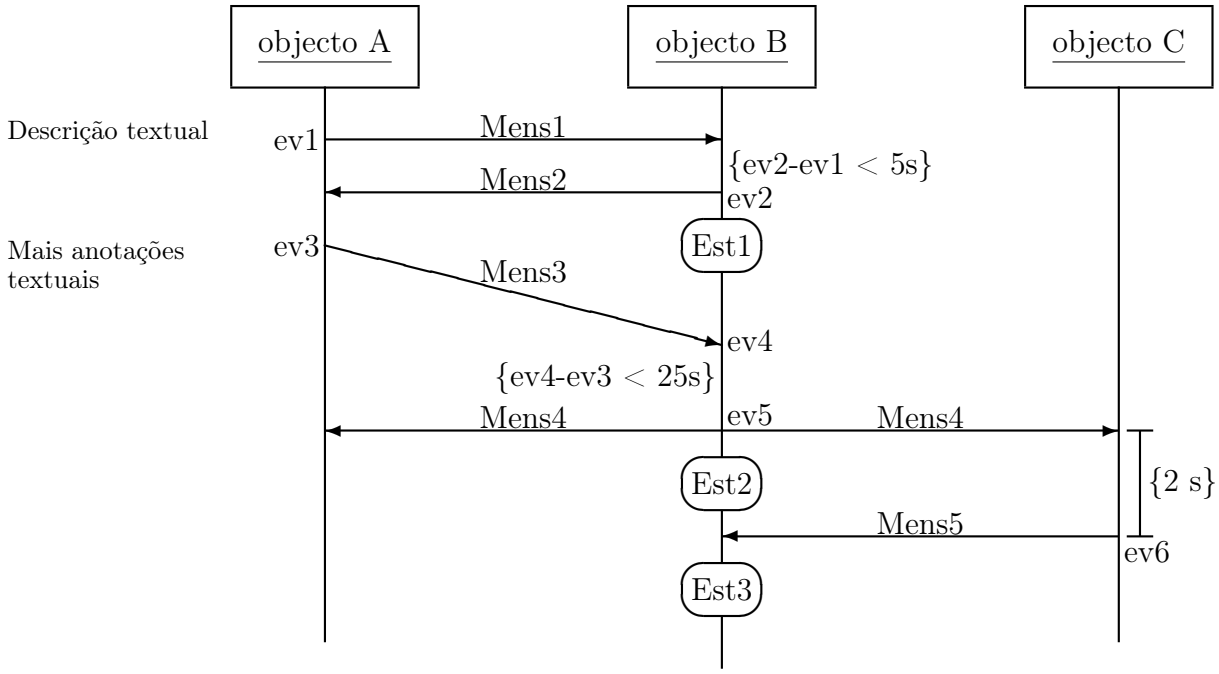


Figura 3.14: Um diagrama de sequência mais elaborado.

5s.

É ainda possível colocar *marcas de estados* nos diagramas de sequência. As marcas de estado, representadas por rectângulos com os cantos arredondados (o mesmo símbolo de estado dos state-charts), são colocadas na linha vertical dum dado objecto, permitindo identificar os vários estados em que esse objecto se pode encontrar. Trata-se dum modo de relacionar, mais facilmente, os diagramas de sequência com os diagramas de estado. Para o objecto B da figura, são identificados três estados distintos: Est1, Est2 e Est3.

Finalmente, mensagens difundidas (*broadcast messages*) são representadas por várias setas a partir do mesmo ponto. Na figura, o objecto B envia uma mensagem difundida (Mens4) para os outros dois objectos. Esta mensagem que ocorre devido ao evento ev2, faz com que o objecto B passe a ter Est2 como o seu novo estado interno.

Diagramas de colaboração

Um *diagrama de colaboração* também pode ser usado para descrever possíveis cenários dum sistema. Os diagramas de colaboração podem ser vistos como diagramas de objectos que apresentam as mensagens que circulem entre eles e por que ordem.

Basicamente, o mesmo tipo de informação é mostrado pelos diagramas de sequência e de colaboração. A diferença reside no facto de os primeiros focaram a sequência de mensagens, enquanto que os últimos enfatizam a estrutura dos objectos que interactuam. Nos diagramas de colaboração, identificar a sequência de mensagens não é tão óbvio como sucede nos diagramas de sequência.

A fig. 3.15 mostra um pequeno exemplo dum diagrama de colaboração, ilustrando o mesmo cenário que foi considerado na fig. 3.13. Os elementos mais comuns que podem ser encontrados neste tipo de diagramas são mostrados na figura.

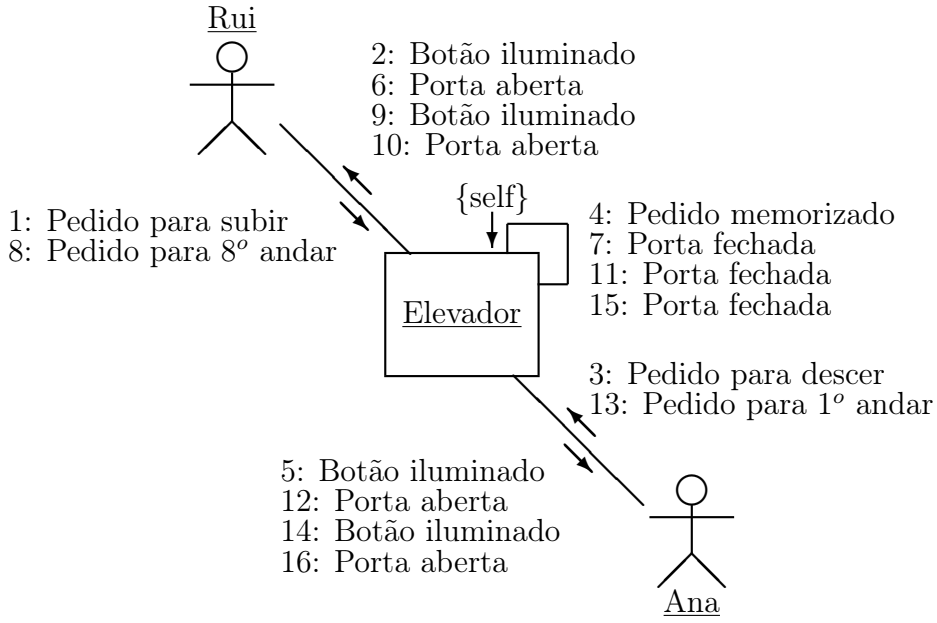


Figura 3.15: Um diagrama de colaboração.

Os objectos que participam no cenário são identificados pelo respectivo nome inserido num rectângulo (mesma notação usada para os objectos em diagramas de objectos) ou por actores (no caso de se tratarem de entidades exteriores) e as ligações mostram quais os objectos que interagem entre si. Dois objectos podem comunicar directamente entre si (i.e. trocar mensagens), apenas se estiverem conectados por uma ligação.

As mensagens são representadas por um identificador, existindo também um número, que precede esse identificador, para definir a ordem das mensagens no cenário. O sentido da mensagem, indicado por uma seta, explicita quais o emissor e o receptor dessa mensagem.

3.3.5 Diagramas de estados

Um diagrama de classes não permite determinar o comportamento dinâmico das instâncias dessas classes, pelo que, para algumas classes mais complexas, o uso de uma notação que tenha subjacente um meta-modelo baseado em estados é crucial. Historicamente, o uso de diagramas de estado, para definir o comportamento dum sistema, popularizou-se no domínio do hardware, mas a sua capacidade de modelação revelou-se útil em diversas disciplinas da Informática (software, compilação, comunicações, sistemas operativos, simulação, multimédia, interfaces homem-máquina, etc.).

Os diagramas de estados são usados para definir o comportamento (dinâmico, temporal) duma classe (i.e. das suas instâncias), explicitando, para tal, todos os estados em que cada um desses objectos se pode encontrar e as transições entre estados provocadas por condições a que esse objecto é sensível. Num diagrama de estados convencional, em cada instante, um e um só estado está activo. Um *estado* dum sistema é representado por um conjunto de variáveis, cujos valores contêm toda a informação necessária sobre o passado do sistema e que, simultaneamente, condicionam o comportamento futuro do sistema. Um estado representa um período de tempo durante o qual o sistema exibe um tipo específico de comportamento. Uma definição de estado apropriada a este trabalho é a seguinte:

> "*A state is an ontological condition that persists for a significant period of time, is distinguishable from other such conditions, and is disjoint with them. A distinguishable state means that it differs from other states in the events it accepts, the transitions it takes as a result of accepting those events, or the actions it performs. A transition is a response to an event that causes a change in state.*" [Douglass, 1998, pág. 155].

Em UML, existem dois formalismos diferentes para especificar máquinas de estados: *state-charts* e *diagramas de actividades* (*activity diagrams*). Os state-charts são usados quando a transição entre estados dispara devido principalmente à ocorrência dum evento significativo. Os diagramas de actividades mostram-se apropriados quando a transição de estados ocorre, sobretudo, por causa da conclusão da actividade executada no estado e não devido à ocorrência de eventos, sejam eles síncronos ou assíncronos. Uma vez que os sistemas de interesse neste trabalho incluem sistemas reactivos (que devem reagir a eventos que ocorrem a qualquer momento), a escolha do formalismo UML para especificação de máquinas de estados recaiu nos state-charts, a exemplo do que sucede em muitas metodologias orientadas ao objecto (como ROOM e OMT).

Os state-charts estendem os diagramas de estados mais convencionais em três vertentes relacionadas com hierarquia, concorrência e comunicação [Harel, 1987]. Apesar destas extensões permitirem obter modelos mais simples, compactos e legíveis, os state-charts são matematicamente equivalentes a máquinas de estados (dos tipos Moore e Mealy) [Douglass, 1999].

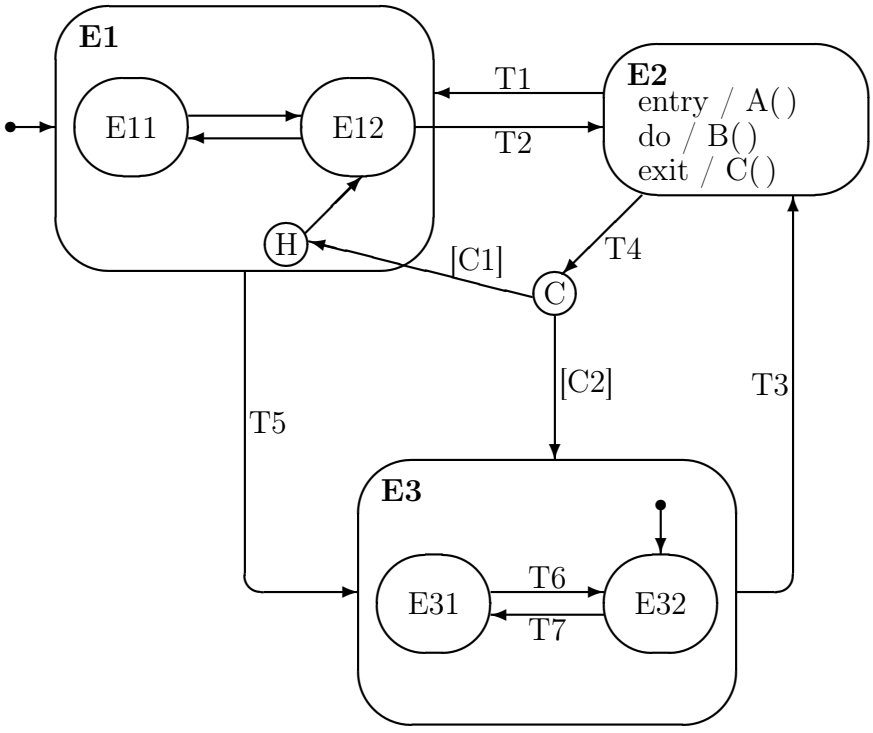


Figura 3.16: Um state-chart.

Um state-chart, de que se mostra um exemplo na fig 3.16, é composto por dois elementos de composição principais, os estados e as transições, representados, respectivamente, por rectângulos com os cantos arredondados e por setas. O diagrama mostra também os super-estados E1 e E3. Um *super-estado* representa um estado que contém outros estados no interior do seu

contorno, sendo o mecanismo usado para estruturar hierarquicamente um state-chart. Este mecanismo de abstracção permite que possa construir-se e ver-se um state-chart ao nível de pormenor desejado.

As transições são usadas para ligar quaisquer dois estados, independentemente dos níveis de profundidade a que estes estejam. Por exemplo, T6 faz com que se transite do estado E31 para o estado E32. No caso das transições T1 e T5, devido ao facto de elas terminarem em super-estados (E1 e E3, respectivamente), deve aplicar-se alguma regra para determinar qual o sub-estado que é activado. Uma hipótese é identificar um dos estados como inicial, usando um círculo negro e uma seta: o estado E32 é o estado inicial do super-estado E3. Outra solução é explicitar um *conector história* (*shallow history state*), como se faz no super-estado E1 relativamente ao sub-estado E12, usando um círculo com um H no interior e uma seta. Neste caso, a transição T1 determina que o sub-estado a activar seja o último que foi abandonado quando se saiu de E1 pela última vez (no caso, E11 ou E12). Trata-se duma forma que permite memorizar o último sub-estado activo que deve ser retomado assim que se reentrar no respectivo super-estado. Se não há história para o super-estado (por ainda não ter estado activo ou por anteriormente ter atingido o seu pseudo-estado final), o conector história aponta para o estado inicial por defeito.

As etiquetas das transições podem ser compostas por três elementos, todos eles opcionais, segundo a seguinte forma: *Evento[Guarda]/Acção*. "Evento" representa o nome do evento que habilita o disparo da transição, "Guarda" é uma expressão booleana que tem de ser avaliada como verdadeira para que a transição dispare e "Acção" é uma lista das operações que são executadas como resultado da transição ser disparada. Nas acções incluem-se, além de comandos simples (como por exemplo a atribuição dum valor a uma variável: cont = 5) ou chamadas a operações, listas com eventos gerados. Esses eventos serão propagados para todos os objectos do sistema que a eles são sensíveis.

Em UML, o termo *evento* tem um âmbito mais alargado do que aquele que lhe está habitualmente associado. Um evento UML representa uma ocorrência cujas consequências são importantes para o sistema em causa. Existem 4 tipos diferentes de eventos em UML:

1. **Evento chamada** (*call event*): Representa um pedido síncrono e explícito dum objecto a outro, esperando o primeiro pela resposta do segundo. É uma forma de invocar a um objecto um seu método, visto como uma implementação duma operação.
2. **Evento transição** (*change event*): Expressa a mudança no valor duma dada expressão booleana (quando passa de falso para verdadeiro). Este tipo de eventos implica um teste em contínuo, podendo, na prática, ser relaxado com análises em determinados instantes, desde que se garanta que não há perda de qualquer ocorrência.
3. **Evento sinal** (*signal event*): Descreve a recepção dum sinal assíncrono (identificado e explícito) para comunicação entre objectos.
4. **Evento temporal** (*time event*): Exprime a chegada do tempo absoluto ou a passagem dum tempo relativo.

Em implementações reais, os eventos temporais não provêm do universo, mas antes de algum objecto (componente) relógio disponível dentro ou fora do sistema. Assim, na prática, os eventos temporais são, na sua génese, semelhantes aos sinais, nomeadamente nos sistemas de tempo-real distribuídos.

Como se viu, quando uma transição termina no contorno dum super-estado, um (e um só) dos seus sub-estados tem de ser activado. Por outro lado, quando a transição se faz a partir do contorno dum super-estado, está-se perante uma simplificação gráfica que representa a aplicação

dessa transição a todos os seus sub-estados. Este último mecanismo, associado com o conector história, é extremamente útil, por exemplo, para explicitar condições de excepção que mereçam tratamento em qualquer situação, devendo o sistema após um dado tratamento retomar o seu estado anterior. O state-chart da fig. 3.17(a) pretende ilustrar a simplificação gráfica que advém da utilização destes dois mecanismos de modelação, para o exemplo considerado, relativamente ao state-chart da fig. 3.17(b), em que esses mecanismos não são utilizados.

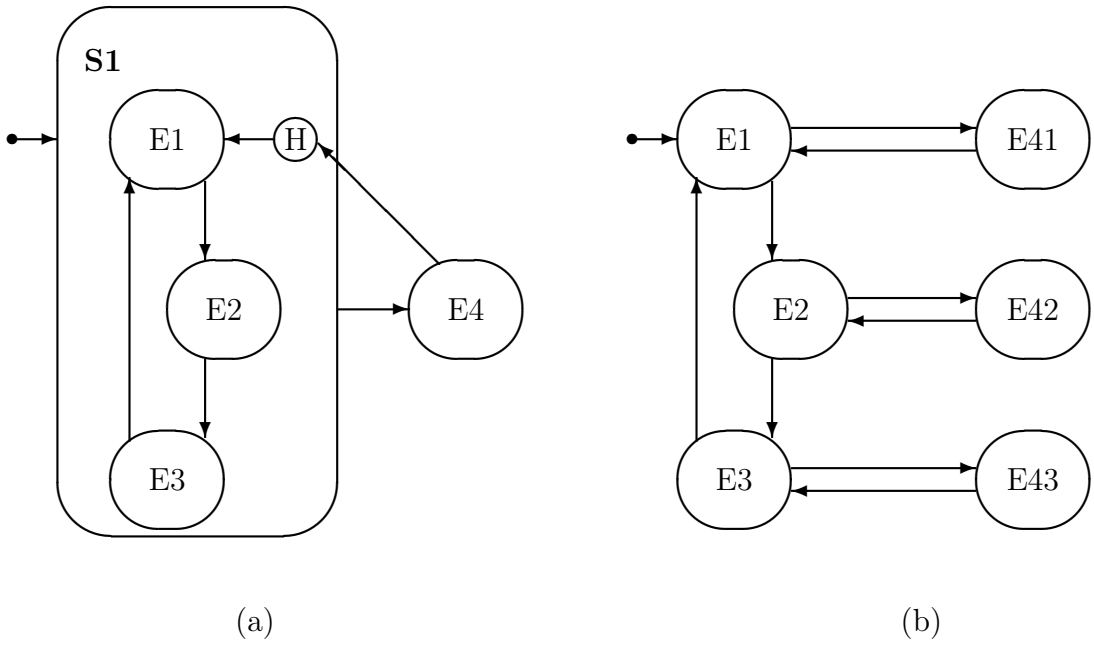


Figura 3.17: Tratamento de condições de excepção, usando transições a partir dum super-estado e o conector história.

Retomando o state-chart da fig 3.16, repare-se na transição T4 que termina num conector condicional (representado por um círculo com um C). Trata-se duma transição cujo estado seguinte depende da avaliação duma condição booleana, no caso as guardas associadas às transições. Se a guarda [C1] for verdadeira, então o estado seguinte é E11 ou E12 (devido ao conector história será o último destes estados que esteve activo na última vez que o super-estado E1 esteve activo). Se a guarda [C2] for verdadeira, então o estado seguinte é E32. Para que o state-chart seja determinístico é necessário garantir que as condições são completas (todas as possibilidades são contempladas) e que são disjuntas entre si (nunca duas guardas ficam simultaneamente habilitadas). O uso da guarda [else] pode ser considerado para garantir estas duas condicionantes.

Considere-se agora a fig 3.18, que será usada para mostrar mais algumas características dos state-charts. O comportamento deste sistema inicia-se no ponto negro (no topo da figura), o que significa que a primeira transição a ser disparada determina que o próximo estado será "verificando". Essa transição determina a execução da acção "procura próximo item". A transição inicial de mais alto nível dum state-chart, que é disparada no momento em que o objecto é criado, não pode ter qualquer evento associado, sendo apenas permitido especificar uma acção que é executada mal o objecto acaba de ser criado. Assim que essa acção terminar, o estado actual do sistema passa então a ser "verificando", que tem associada a actividade "verifica item".

Em UML, distingue-se entre acção e actividade. Ambas representam processos que têm de ser executados e são, tipicamente, operações disponibilizadas pelo objecto em causa ou pelos

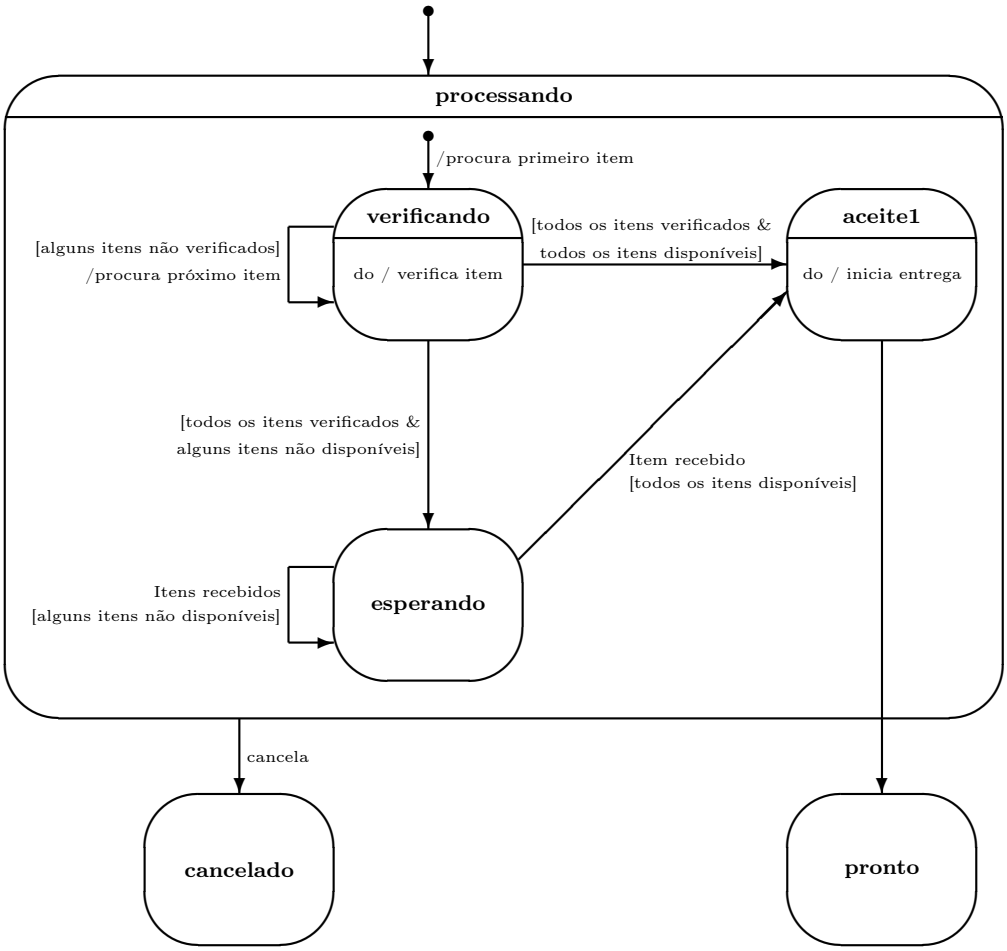


Figura 3.18: Um outro state-chart.

objectos que com ele comunicam. Uma *acção* está associada a uma transição e é considerada como um processo instantâneo, no sentido de ocorrer muito rapidamente e de não ser susceptível de interrupção. Uma *actividade* está associada a estados e normalmente será uma operação mais demorada, o que determina que possa ser interrompida. Aos estados podem associar-se acções de entrada e saída (*entry actions* e *exit actions*), bem como actividades. As acções de entrada são operações que se realizam quando se entra no estado, e as acções de saída são realizadas quando se sai desse estado. As actividades são realizadas enquanto o estado está activo (são indicadas pela palavra *do*). No state-chart da fig 3.16, o estado E2 tem estes 3 tipos de operações.

Os state-charts podem ser construídos duma forma incremental e, para exemplificar esta possibilidade, considere-se o state-chart da fig 3.19 que representa um refinamento do state-chart da fig 3.18. Por questões de simplicidade, omitiram-se as etiquetas das transições. O comportamento deste sistema inicia-se no ponto negro (do lado esquerdo da figura), o que significa que o primeiro estado activo é o super-estado "processando". A linha a tracejado separa duas regiões ortogonais, que representam dois comportamentos independentes, introduzindo assim concorrência no modelo do sistema.

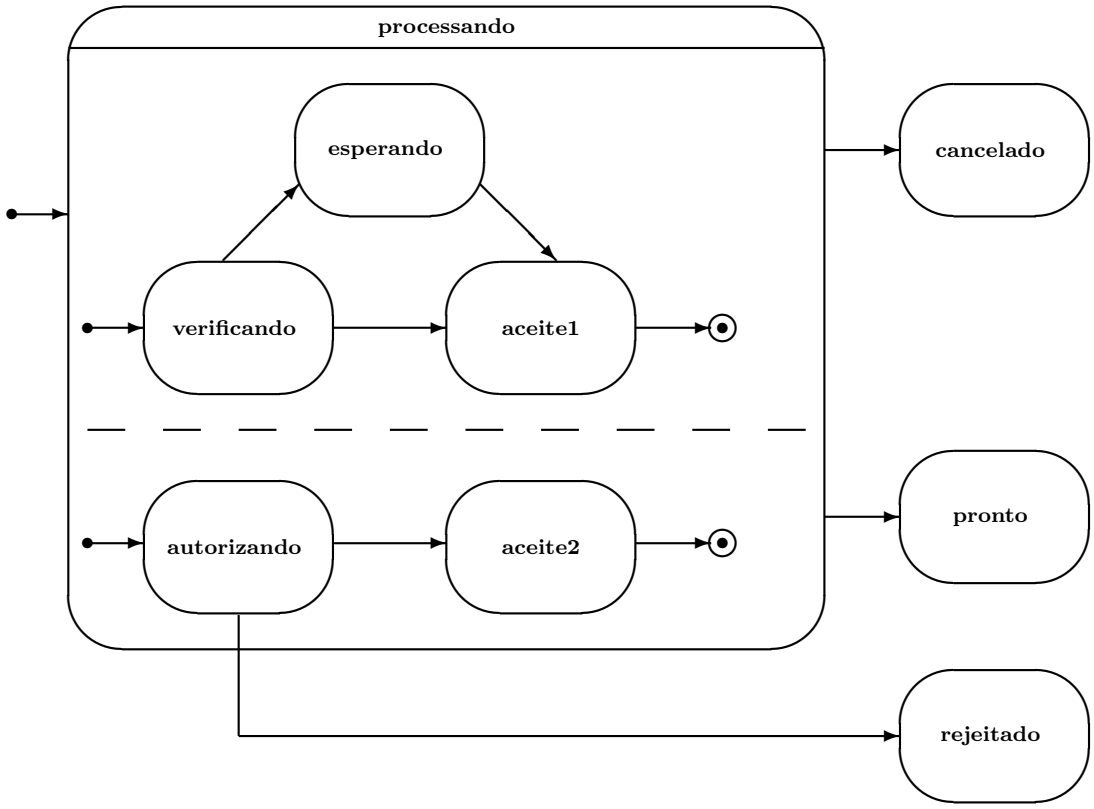


Figura 3.19: Um state-chart com concorrência.

O objecto quando está no estado "processando", tem de estar obrigatoriamente em exactamente dois sub-estados, um de cada uma das suas regiões (por exemplo, "verificando" e "autorizando"). Embora esta modelação não implique necessariamente concorrência entre fios de execução, tal mecanismo consiste numa das soluções de implementação para esses casos. As regiões ortogonais podem comunicar entre si, por intermédio de eventos ou guardas dependentes de variáveis comuns, de modo a sincronizar os seus comportamentos.

Informação mais completas sobre state-charts, nomeadamente alguns mecanismos de modelação não apresentados aqui, podem ser obtidas em [Harel, 1987] [Harel e Gery, 1997] [Douglass, 1998] [Douglass, 1999] [Rumbaugh et al., 1999, cap. 6].

3.3.6 Outros mecanismos

A definição de UML baseou-se, em forte medida, nas notações associadas às metodologias de Booch, OBJECTORY e OMT. Houve, na sua definição, a preocupação de consolidar essas notações, mas também de introduzir alguns mecanismos avançados de modelação. Nesse sentido, foram adicionados a UML, entre outros, os estereótipos, os valores etiquetados e as restrições. Estes conceitos unificam uma série de características das notações daquelas três metodologias e permitem estender a notação UML [Booch, 1996, pág. 105] [Booch et al., 1999, pág. 75].

Notas de texto

Uma *nota de texto* (*text note*) é um elemento gráfico que pode ser colocado em qualquer tipo de diagrama e que não tem qualquer valor semântico (fig. 3.20). A sua utilização reduz-se à colocação de comentários e notas para aumentar a legibilidade dos modelos. Por exemplo, relativamente a uma classe, é conveniente introduzir uma nota de texto onde se indicam comentários relativos àquela, incluindo, por exemplo, responsabilidades, propósito, restrições (pré e pós-condições) e regras.

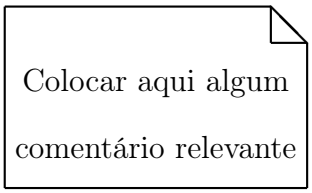


Figura 3.20: Uma nota de texto.

Estereótipos

Um *estereótipo* (*stereotype*) é um mecanismo que permite estender a notação UML, por forma a incluir elementos não previstos na notação base e é representado por texto entre aspas, como por exemplo ≪extends≫, ou por um símbolo gráfico especial que a equipa de projecto define, como, por exemplo, o ícone para actores. Os estereótipos permitem rotular (etiquetar) uma dada classe, tratando-se duma forma de classificar as classes (meta-classificação)[13]. Cada classe pode ter associados zero ou mais estereótipos.

Os estereótipos duma classe são listados por cima do nome da classe, servindo as aspas para distinguir os estereótipos do nome da classe. A utilização de cores ou tipos de letra para distinguir diferentes estereótipos é aconselhada, nomeadamente quando se usarem utilitários para edição electrónica [Booch, 1996, pág. 125–6]. Alternativamente, podem adoptar-se símbolos

[13]Um estereótipo não é uma meta-classe. Uma *meta-classe* é a classe duma classe e trata-se dum conceito importante, por exemplo, em SMALLTALK.

especiais, mas a sua introdução deve ser feita com algum critério, sobretudo se existirem utilitários comercialmente disponíveis, pois pode revelar-se nefasta relativamente à portabilidade do modelo.

Todos os mecanismos de modelação definidos em UML podem ser estereotipados, sendo, por ventura, o exemplo mais óbvio as mensagens. Os estereótipos são usados para indicar, por exemplo, se uma mensagem é síncrona ou assíncrona, se é periódica ou episódica, se o seu fluxo é contínuo ou discreto, etc.

Valores etiquetados

Um *valor etiquetado* (*tagged value*) é uma par (etiqueta,valor) que se associa a qualquer elemento de modelação para guardar informação. Os valores etiquetados constituem uma forma simples de estender o meta-modelo UML e podem ser usados, por exemplo, para passar informação aos utilitários de geração automática de código. Neste sentido, trata-se duma possibilidade a explorar no desenvolvimento de sistemas embebidos (especialmente nas fases de concepção e implementação), sobretudo durante a partição hardware/software, uma vez que permite identificar quais os objectos a implementar em software e quais aqueles a implementar em hardware.

Cada valor etiquetado é composto por uma *etiqueta* (*tag*) e por um valor, com a seguinte forma: *etiqueta* = *valor*. Por exemplo, o seguinte valor etiquetado pode ser usado para indicar o autor dum dado diagrama: {autor = Miguel}.

Restrições

Uma *restrição* é uma condição adicional que se aplica a um dado elemento de modelação e é sempre indicada entre chavetas {}. Por exemplo, a restrição {abstract} é usada numa classe para indicar que se trata duma classe abstracta, enquanto que a restrição {instantiable} é usada para indicar que a classe é concreta. Nos diagramas de sequência, podem usar-se restrições para explicitar marcas temporais (fig. 3.14).

Pacotes

Para sistemas muito complexos e devido às limitações que a mente humana apresenta ao lidar com a complexidade, é impossível tratar todos os elementos de modelação ao mesmo nível. Nesse sentido, é necessário recorrer ao *pacote* (*package*) que é um mecanismo genérico para organizar elementos de modelação em grupos. Um pacote pode conter quaisquer elementos de modelação UML: classes, objectos, componentes, nodos, casos de uso, diagramas e até outros pacotes. A possibilidade de um pacote pode conter outros pacotes permite organizar hierarquicamente os modelos em causa, tratando-se dum mecanismo extremamente útil para sistemas de elevada complexidade.

A noção de sistema ou subsistema é incluída em UML, criando-se um pacote estereotipado com ≪system≫ e ≪subsystem≫, respectivamente.

Tipicamente, o uso de pacotes e subsistemas mostra-se necessário, quando um dado diagrama se apresenta muito complexo, uma vez que permite mostrar os pacotes em vez dos elementos que os constituem [Booch et al., 1999, pág. 103]. Graficamente, um pacote pode representar-se

segundo qualquer das formas indicadas na fig. 3.21. O pacote "Base Dados" por apresentar as classes que o constituem tem o respectivo nome indicado no rectângulo da parte superior esquerda.

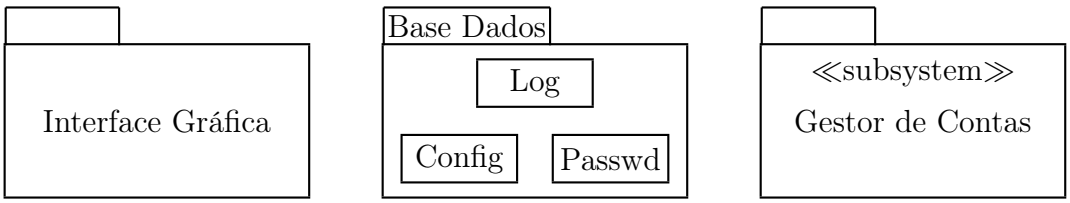


Figura 3.21: Notação gráfica para pacotes.

3.4 Utilitários para desenvolvimento orientado ao objecto

O ser humano, desde os tempos primitivos, que constrói e usa ferramentas, para tornar mais simples ou, pelo menos, mais cómodas as tarefas que tem de realizar. No desenvolvimento de sistemas, por se tratar duma tarefa complexa, também tem sido constante a procura de utilitários que possam ajudar os projectistas. Os utilitários podem ser usados para automatizar um elevado número de tarefas, sobretudo aquelas que são triviais ou repetitivas, mas que requerem muito tempo para a sua realização.

Tradicionalmente, o desenvolvimento de programas requeria alguns utilitários: um editor de texto, um compilador, um *linker*, um *loader* e eventualmente um depurador (*debugger*). O desenvolvimento orientado ao objecto, para se tornar exequível, necessita dum conjunto mais alargado de utilitários, nomeadamente editores gráficos para desenhar os diversos diagramas necessários a um dado projecto.

Embora seja possível desenvolver programas orientados ao objecto, sem usar linguagens que também o sejam, assume-se por motivos pragmáticos, durante esta secção, que tal não sucede.

Desenhar modelos gráficos em papel é uma tarefa que consome muito tempo e onde é susceptível serem cometidos muitos erros, sobretudo, se for necessário proceder a frequentes alterações. Assim, é necessário um editor que suporte a notação gráfica usada para representar os sistemas orientados ao objecto. Esse editor gráfico deverá garantir o cumprimento das convenções notacionais, oferecendo assim um nível de suporte ao desenvolvimento de software muito idêntico àquele que é dado por ferramentas CAD noutras áreas (arquitectura, mecânica, calçado, etc.).

Sendo o conceito de classe considerado nuclear em qualquer desenvolvimento orientado ao objecto, é necessária a existência dum navegador de classes (*browser*) que facilite a pesquisa, consulta e alteração das classes que compõem a hierarquia de classes. Quando a estrutura de classes ainda é relativamente pequena, este utilitário não se revela fundamental, mas, assim que o tamanho da estrutura se torna apreciável é completamente indispensável a sua existência. Repare-se que a necessidade do navegador não depende da complexidade do sistema em desenvolvimento, mas sim da grandeza da hierarquia de classes (i.e. do número de classes).

Apesar das promessas e expectativas que se foram criando em torno das potencialidades que os utilitários CASE poderiam oferecer (chegou mesmo a aventar-se a hipótese de que eles substituiriam o ser humano, em todas as fases), a realidade é bem mais amarga. A prática tem mostrado que os utilitários CASE tiveram um contributo significativo, sobretudo, na autom-

atização de duas actividades (a documentação do projecto e a geração de código), não tendo, todavia, sido tão bem sucedidos relativamente a outras tarefas.

Um ambiente CASE deve oferecer um repositório que consiste essencialmente numa base de dados ligada aos utilitários de desenho. Esse repositório deve permitir que seja listada, segundo uma dada ordem (alfabética, por exemplo), a informação relativa a qualquer objecto ou classe de interesse para o projectista, mostrando todas as características desse elemento de modelação (associações, atributos, operações, responsabilidades, estrutura, state-chart, etc.). Esta possibilidade permite reunir, num único elemento de projecto, várias perspectivas de modelação relacionadas, mas expressas em diagramas distintos.

A geração de código, a partir dos modelos gráficos, é uma das características mais publicitadas dos utilitários CASE, por ser provavelmente aquela que está mais bem consolidada, permitindo libertar a equipa de projecto dessa tarefa mais ou menos mecânica. No entanto, há algumas considerações que devem ser feitas relativamente a esta facilidade. Uma primeira refere-se à evolução dos modelos gráficos e a sua repercussão com o código gerado e, neste sentido, um gerador de código é de utilização duvidosa se não fizer reflectir automaticamente no código as alterações que se fizerem nos modelos de análise ou concepção. O segundo reparo é a relação dos utilitários CASE com a fase de depuração do código, depois deste ter sido gerado. Se a depuração tiver de fazer-se directamente sobre o código, sem qualquer relação automática com os modelos que estiveram na sua origem, pode ser difícil estabelecer mentalmente essa relação, diminuindo assim a utilidade desta facilidade.

3.5 Resumo final

Neste capítulo, sugeriu-se uma definição do termo objecto, no âmbito do desenvolvimento de sistemas segundo o paradigma dos objectos. A visão dos objectos como tipos abstractos de dados é demasiado limitadora para sistemas complexos. Daí que, neste trabalho, um objecto seja visto como uma máquina (agente computacional), no sentido de ser um componente dum sistema computacional. Esta perspectiva, não sendo contraditória com o conceito de tipo abstracto de dados, é mais genérica e abrangente, pois enquanto que uma instância dum tipo abstracto de dados está sempre num estado passivo e "acorda" sempre que uma das suas operações é invocada, um agente computacional, por ter o seu próprio fio de controlo, pode estar activo durante um largo período de tempo. Fez-se ainda a distinção entre o termo "orientado ao objecto" e os termos "baseado em classes" e "baseado em objectos".

Foram também indicadas várias características e propriedades associadas ao paradigma dos objectos, nomeadamente a identidade, a classificação, a abstracção, o encapsulamento, as mensagens, o polimorfismo, a herança e a agregação. Todos estes mecanismos de modelação auxiliam o projectista a lidar com a complexidade dos sistemas.

Foi feita uma descrição dos vários elementos que compõem a notação UML, que pode ser usada para modelar, segundo a abordagem orientada ao objecto, as várias vistas dum sistema complexo. Foram apresentados os diagramas de casos de uso, os diagramas de classes, os diagramas de objectos, os diagramas de interacção, os diagramas de estados e outros mecanismos (notas de texto, estereótipos, valores etiquetados, restrições e pacotes) disponibilizados por UML.

Concluiu-se o capítulo, com uma referência a algumas questões relacionadas com os utilitários que têm de existir para o desenvolvimento orientado ao objecto ser exequível e eficiente.

Parte II

Contributo científico

Capítulo 4

A Metodologia MIDAS

Mais faz quem quer do que quem pode.

Sumário

Neste capítulo, tenta justificar-se a necessidade duma metodologia orientada ao objecto para desenvolvimento de sistemas embebidos, apresentando primeiramente a utilidade duma metodologia e depois as vantagens que decorrem se ela seguir o paradigma dos objectos em contraponto à abordagem estruturada. De seguida, indicam-se os elementos que foram considerados nucleares na definição da metodologia MIDAS*, nomeadamente o uso da abordagem operacional, de especificações gráficas e multi-vista, e de modelos orientados ao objecto. São também apresentados alguns considerandos genéricos relativamente às fases de análise, concepção e implementação, no âmbito da metodologia* MIDAS*, de modo a dar uma perspectiva global desta. O capítulo é concluído com uma referência aos principais contributos que influenciaram a elaboração da metodologia* MIDAS*.*

Índice

4.1 Uma metodologia para sistemas embebidos

Há umas décadas, o sistema embebido típico a desenvolver era relativamente simples, sendo portanto o seu comportamento intuitivamente óbvio ou, quanto muito, esquematizado em meia dúzia de folhas de papel. A equipa de projecto, após uma análise breve dos requisitos do sistema, começava o seu desenvolvimento, codificando directamente na linguagem de programação escolhida. Contudo, nos últimos anos, a complexidade dos sistemas digitais cresceu quatro vezes em cada triénio. Este crescimento, observado em 1965 por Gordon Moore, ficou conhecido como a lei de Moore [Moore, 1965][1] e tem-se verificado até aos dias de hoje, sendo de esperar que continue a sê-lo no futuro [Ecker e Mrva, 1996], razão pela qual a abordagem seguida para desenvolver sistemas embebidos tem, obrigatoriamente, de ser substancialmente alterada.

Os engenheiros têm de estar conscientes de que, cada vez que a escala varia, aparecem problemas diferentes [Mota, 1999]. Para se perceber a necessidade duma metodologia orientada ao objecto para desenvolver sistemas embebidos complexos, atente-se na seguinte afirmação, feita em 1987, em relação a esta questão:

> "*Most systems are still modeled with the same flow charts and specification documents that have been used since the 1950s. But it has become obvious that, with the increasing complexity, today's design techniques are inadequate in the face of large, event-driven multiprocessor designs. Consequently, the future will bring new tools. Whether these will be based on existing data-flow models or new object-oriented design techniques isn't clear, but in any case the new tools will rely on concepts unfamiliar to most of today's designers. To remain competitive, designers will have to learn not only new technologies, but also new fundamental concepts in system design.*" [Wilson, 1987].

O autor deste trabalho é também da opinião que é fundamental, para o bom sucesso na realização dum sistema embebido de elevada complexidade, que se adoptem metodologias de desenvolvimento em que nenhuma das fases pré-implementação seja vagamente considerada ou mesmo esquecida. Em particular, o levantamento de requisitos e a análise têm de ser consideradas actividades cruciais, o que, em muitas situações, não sucede. Nesse sentido, e tendo em consideração as razões apontadas nas secções 2.3.1 e 2.3.2, a adopção de metodologias orientadas ao objecto parece permitir desenvolver, com algum sucesso e dum modo sustentado, sistemas de software de grande complexidade, pelo que será legítimo considerar que resultados idênticos se podem obter no desenvolvimento de sistemas embebidos. A necessidade de metodologias depende, em forte medida, da complexidade dos sistemas e não tanto do seu tipo.

É importante que a metodologia usada na construção dum sistema seja devidamente valorizada. Se bem que o valor e a qualidade do produto final constituam um factor determinante, não se deve desprezar o processo que foi seguido para o obter. O cliente interessa-se apenas pelos primeiros, mas o projectista tem que dar igual importância a ambos. O crescente interesse pelos sistemas e normas de qualidade nas várias actividades industriais vem mostrar que a garantia do processo de fabrico é cada vez mais uma necessidade das organizações que desenvolvem qualquer produto ou serviço. É de supor que idêntico interesse se venha a verificar, num futuro não distante, em instituições que vendem soluções de software e de hardware.

[1] No URL `www.intel.com/intel/museum/25anniv/hof/moore.htm` podem obter-se mais informações sobre a lei de Moore.

É sabido que, na prática, o aspecto mais difícil de modificar em qualquer instituição é o seu *modus-faciendi*, ou seja, a forma como actua perante os problemas que lhe são colocados para resolução. Quando confrontado ante uma nova estratégia, para substituir uma outra já anteriormente usada com relativo êxito, o ser humano, mesmo que perante promessas de mais vantagens, é, normalmente, muito renitente em adoptar a nova proposta. Exemplificando, quando alguém é colocado perante a possibilidade de trocar o processador electrónico de texto, que já utilizou com sucesso e que conhece razoavelmente bem, por um outro mais poderoso, mas que nunca utilizou, a resposta é, muitas vezes, equivalente à seguinte: "*Tudo bem, é capaz de ser melhor, mas o que eu uso funciona bem e eu não necessito de nada mais poderoso!*".

Este tipo de resposta vulgariza-se, quando se pretende alterar toda uma metodologia de desenvolvimento, uma vez que a inércia à mudança se acentua ainda mais, pois as alterações são mais profundas, o que implica a aprendizagem, não só de novos utilitários, como também de novas notações, novas linguagens, novos conceitos e novas atitudes, sem a certeza de daí se obterem vantagens reais com essa alteração.

Apesar desta realidade e de se reconhecerem as inúmeras dificuldades em introduzir, em ambientes industriais, a metodologia proposta neste trabalho, é função dos agentes académicos pôr em causa os processos habitualmente usados, experimentar novas metodologias, alterar mentalidades, e propor novas técnicas e abordagens, pelo que se justifica plenamente, no entender do autor, a realização desta tese de doutoramento.

As metodologias orientadas ao objecto para desenvolvimento de sistemas complexos são o resultado duma longa e vasta experiência em inúmeros projectos realizados em grandes companhias mundiais, abertas à introdução de novas tecnologias, novas ideias e novas formas de actuar. Essa experiência acumulada de soluções e sucessos, mas igualmente de problemas, erros e falhas, permitiu que os seus proponentes fizessem uma retrospectiva ao desenvolvimento seguido e pudessem, gradualmente, criar uma série de recomendações para projectos futuros. Depois de ter sentido alguns problemas com casos práticos, torna-se mais simples indicar qual o modelo de processo e quais os métodos para as várias fases (i.e. a metodologia) que melhor se adequam ao tipo de sistemas considerado.

Segundo o autor, a definição de novas metodologias orientadas ao objecto para desenvolvimento de sistemas embebidos deve pois ser incentivada. Porém, não se deve aqui cometer um erro muito comum: inventar de novo o que já existe (aquilo a que habitualmente se chama "reinventar a roda"). Assim, as metodologias orientadas ao objecto para desenvolvimento de software devem ser devidamente estudadas e todo o capital acumulado de resultados (positivos ou negativos) deve ser reaproveitado, de modo a adoptar essas metodologias, com as devidas alterações, tendo em conta as especificidades e as características próprias dos sistemas embebidos.

O papel do autor é, pois, definir uma metodologia que congregue as características mais pertinentes das metodologias existentes, tendo em conta os objectivos deste trabalho (secção 1.4). A metodologia a definir deverá, de alguma forma, ser construída e adaptada, tendo como pano de fundo a realidade portuguesa[2]. As metodologias mais populares foram definidas em grandes companhias com sede em países desenvolvidos (E.U.A., Reino Unido, França, Japão)[3], onde

[2]Não se pretende, de forma alguma, limitar o uso da metodologia exclusivamente ao contexto português, mas tão somente tentar elaborá-la no pressuposto que será esse um dos mercados preferenciais a explorar.

[3]OMT [Rumbaugh et al., 1991] nasceu no centro de investigação e desenvolvimento da General Electric (E.U.A.), HOOD [Robinson, 1992] surgiu na agência espacial europeia (ESA) e a metodologia de Booch [Booch, 1991] foi já usada em projectos para companhias como, por exemplo, AT&T, Boeing, Ericsson, IBM, Lockheed, Philips e Shell.

existe, certamente, uma cultura tecnológica bem diferente daquela que se encontra no contexto português, pelo que propor ou usar directamente uma metodologia, sem as devidas precauções, pode revelar-se um verdadeiro fracasso. A metodologia deve ainda ser devidamente validada, de forma a mostrar as suas capacidades (e debilidades) na modelação de sistemas embebidos. Neste sentido, a escolha de sistemas da indústria portuguesa, como casos de estudo, permitirá mais facilmente adaptar a metodologia à realidade nacional.

Refira-se que, actualmente, uma metodologia só tem a aceitação da comunidade científica e dos seus potenciais utilizadores, se identificar as fases de desenvolvimento e os seus fins e se sugerir uma dada ordem para a sua aplicação, mas permitindo a operação simultânea e a interacção entre fases [Morris et al., 1996, pág. 25].

Em jeito de conclusão, pode afirmar-se que uma metodologia não deve ser vista como um receituário milagroso, como uma panaceia para todos os problemas, que, se seguida sem desvios, produz sempre os melhores resultados. Para desenvolver sistemas complexos, não pode recorrer-se a abordagens do tipo "livro de receitas", visto que esse tipo de sistemas, dada a sua natureza, tem necessariamente associado um processo de desenvolvimento incremental e iterativo [Booch, 1991, pág. 21]. Além do mais, o responsável pelo desenvolvimento do sistema é, em última análise, o projectista, que deve ter necessariamente uma atitude crítica, na medida do possível, em relação aos métodos e ao modelo do processo que utiliza. Uma metodologia, qualquer que ela seja, não se adapta igualmente bem a todos os tipos de sistemas, pelo que segui-la cegamente, além de inibir fortemente a capacidade criativa do projectista, pode resultar em custos de desenvolvimento agravados. Neste sentido, tornam-se muito atractivos os utilitários CASE, independentes da metodologia (modelo do processo e dos métodos para as várias fases), por possibilitarem a sua adopção a um leque alargado de metodologias e sistemas.

4.2 Caracterização da metodologia

Nesta secção, apresentam-se algumas das características mais importantes da metodologia para o desenvolvimento de sistemas complexos, proposta no âmbito deste trabalho e que se designou por MIDAS (**M**etodologia **I**ntegrada para **D**esenvolvimento de sistem**AS**). A metodologia MIDAS foi idealizada em torno dos seguintes elementos, que foram considerados nucleares na sua definição:

- Abordagem operacional.
- Especificação unificada, gráfica e multi-vista.
- Modelação orientada ao objecto.

A abordagem operacional usa modelos de especificação que podem ser vistos como programas escritos numa linguagem de alto nível. Esses programas são compilados e executados, a fim de se detectarem, durante o desenvolvimento do sistema, falhas e inconsistências.

A metodologia usa uma abordagem baseada numa representação unificada e neutra, de que resultam as vantagens indicadas na subsecção 2.5.2. A representação do sistema vai sendo progressivamente refinada, até se obter um modelo válido do sistema. Nessa altura, podem então escolher-se as tecnologias, nas quais os vários componentes vão ser implementados.

As notações para especificar sistemas reactivos devem ter, na opinião do autor, um forte cariz visual e intuitivo, de modo a facilitar a transição da análise para a implementação e a comunicação entre clientes e projectistas, e possibilitando ainda modificações pós-desenvolvimento. Estas necessidades, aliadas à precisão e ao rigor pretendidos, levam à noção de *formalismo vi-*

sual, ou seja, uma notação gráfica, usando preferencialmente um pequeno número de elementos gráficos, mas com uma semântica precisa e bem definida.

O uso de representações gráficas mostra-se útil, sempre que as especificações não estiverem sobrecarregadas com demasiados pormenores[4]. Quando isso não sucede, a legibilidade da especificação é reduzida, dada a inúmera quantidade de elementos que devem ser captados e relacionados. É preferível, nessas circunstâncias, o uso de modelos multi-vistas, cada um dos quais abordando uma dada perspectiva de modelação do sistema.

Neste sentido, adoptar-se-á a linguagem UML, que além de preencher os requisitos atrás indicados (unificada, multi-vista e gráfica), é também a notação oficial do OMG para descrição de sistemas complexos. Apesar de UML ser intencionalmente independente do processo, os mecanismos de modelação que disponibiliza adaptam-se perfeitamente a um processo incremental e iterativo que se baseie em casos de uso e num modelo de objectos [Booch, 1996, pág. 110].

Pretende adoptar-se um modelo de processo "sensato", ou, se se preferir, "realista", no sentido de contemplar a possibilidade de serem cometidos erros em qualquer das fases que o compõem. Assim, um modelo que atenda esta consideração pode ser recuado no seu fluxo, permitindo que sejam revisitadas fases anteriores, com o intuito de rectificar o que possa não estar correcto. Adicionalmente, um modelo não deve ter um fluxo rígido, pois a flexibilidade na aplicação das diversas etapas e dos diversos passos do processo, deve ser entendida como uma característica positiva importante, na medida em que poderá permitir a sua aceitação por um maior número de equipas de projecto, bem como facilitar a incorporação e a adaptação de outras vivências, práticas e culturas metodológicas. Nesta linha de raciocínio, a utilização do modelo em cascata (ou de qualquer uma das suas variantes) será preterida em detrimento de modelos mais iterativos, como o modelo transformacional ou em espiral.

Na abordagem tradicional (secção 2.1), a maior fatia do teste está concentrada no fim do projecto. Para sistemas embebidos, é crucial validar as restrições que condicionam o seu desenvolvimento o mais cedo possível, visto que os custos de desenvolvimento e teste deste tipo de sistemas são elevados. Para construir um sistema que tem de obedecer a um conjunto de requisitos funcionais e restrições não funcionais (por exemplo, de carácter temporal para sistemas de tempo-real), devem construir-se vários modelos de especificação e realizar uma implementação. Ora, após estarem definidos o conjunto de restrições impostas e o modelo de concepção que suporte a funcionalidade pretendida para o sistema, é provável que não seja possível encontrar uma implementação que satisfaça a totalidade das restrições. Neste sentido, um modelo de processo iterativo é novamente importante para o desenvolvimento de sistemas embebidos, já que não impõe a necessidade de finalizar a implementação para validar o sistema, o que implicaria um custo muito elevado, no caso de ser necessário proceder a alterações directamente na implementação. Além disso, a utilização de protótipos e a validação dos modelos desenvolvidos, em fases precoces do projecto, permite detectar, estudar e corrigir problemas que, noutras situações, só o poderiam ser em fases finais do projecto (ou, pior ainda, durante a utilização do sistema).

A necessidade de modelos de processo flexíveis é, na prática, sentida, porque, por vezes, não é possível aplicar as diversas fases dum dado processo pela ordem em que ele foi inicialmente idealizado. Por exemplo, os testes de unidade e de integração não são, duma maneira geral, realizados estritamente pela ordem cronológica indicada. Em algumas situações, é desejável

[4]Este é um dos principais inconvenientes, por exemplo, do meta-modelo RdP-shobi, que inviabiliza a sua utilização, para desenvolvimento de sistemas complexos, mesmo considerando a possibilidade de usar mecanismos de hierarquia.

testar um dado conjunto de unidades, integrá-las e testar o resultado e, depois, testar mais unidades e integrá-las na versão corrente do código. Para utilizar um dado modelo de processo, num ambiente longe do ideal, é necessário captar os seus aspectos teóricos e saber contornar as dificuldades, com o cuidado de não ferir os seus princípios fundamentais.

É contudo fundamental que o leitor se aperceba que a metodologia, *per si*, não é suficiente para assegurar êxito no desenvolvimento de sistemas. Ela deve ser adaptada às necessidades específicas da organização e estendida com outras técnicas, que se possam achar, úteis.

A metodologia MIDAS foi concebida para o desenvolvimento de sistemas embebidos, conforme definição dada na secção 1.1. É um pouco ingrato tentar definir ou restringir outras características associadas à metodologia, pois pretende-se que esta seja o mais abrangente possível, mas é óbvio que não são cobertos todos os casos de desenvolvimento de sistemas. Assim, tentará, de seguida, tipificar-se as características e as situações em que a metodologia MIDAS pode ser melhor aproveitada ou nas quais pode ser facilmente enquadrada.

Em princípio, a equipa de projecto será composta por várias pessoas, uma vez que uma pessoa não consegue, sozinha e em tempo útil, desenvolver um sistema complexo. Deste facto resulta obrigatoriamente a necessidade de algum tipo de gestão do projecto (gestão de recursos humanos, gestão financeira, calendarização de tarefas, organização do trabalho em grupos, condução de reuniões, formato da documentação, etc.), que não será considerada neste trabalho, por não se enquadrar nos objectivos do mesmo. De igual forma, a realização dum estudo de viabilidade, antes de se iniciar a fase de análise propriamente dita, revela-se, muitas vezes, de extrema importância para as fases seguintes de projecto, mas também não será encarada, por igualmente não encaixar nos objectivos deste trabalho.

Apesar de ser possível projectar sistemas para uso próprio, a situação mais comum é o sistema ser desenvolvido para outrém. A metodologia MIDAS parte do pressuposto que a equipa de projecto desenvolve um sistema que lhe foi encomendado por um cliente, entendido como uma entidade externa à equipa de projecto (por exemplo, dentro da mesma organização, a equipa de projecto pode pertencer a um departamento diferente daquele donde partiu o pedido para iniciar o projecto). Regra geral, o cliente é especialista no domínio de aplicação, mas não é conhecedor de linguagens de especificação, modelação de sistemas, hardware, software, etc. Por outro lado, dados os inúmeros domínios possíveis em que se podem enquadrar os sistemas a desenvolver, é lícito supor que o conhecimento dos projectistas na área de aplicação é geralmente diminuto: apenas o conhecimento médio (dir-se-ia superficial) de um cidadão atento em áreas de que não é especialista ou especial interessado.

A metodologia MIDAS pode ser usada para desenvolver de raiz sistemas, ou então em situações de engenharia reversa (reengenharia). A primeira situação pode ser considerada um caso especial da segunda, em que se desenvolve o sistema pela primeira vez. A utilização da metodologia MIDAS numa perspectiva de reengenharia levanta algumas questões, embora seja conhecido que os sistemas desenvolvidos segundo a abordagem orientada ao objecto podem, com mais facilidade, ser alterados, o que parece oferecer algum fundamento e dar algumas garantias, quanto à possibilidade da metodologia se mostrar também útil em situações de reengenharia.

4.2.1 O modelo de processo

Apresenta-se, na fig. 4.1, o modelo de processo associado à metodologia MIDAS. Por questões de simplicidade conceptual, o modelo de processo da metodologia MIDAS é apresentado duma

forma sequencial. Há contudo a noção que, na realidade, a sua aplicação seguirá um modelo mais iterativo (com possíveis ligações para trás). Tipicamente, o processo é sequencial no início do desenvolvimento (estudo de viabilidade e análise) mas, mais à frente, torna-se uma mistura de actividades (por exemplo, a reutilização dum objecto, a partir da instanciação duma classe transforma imediatamente um objecto analisado em codificado).

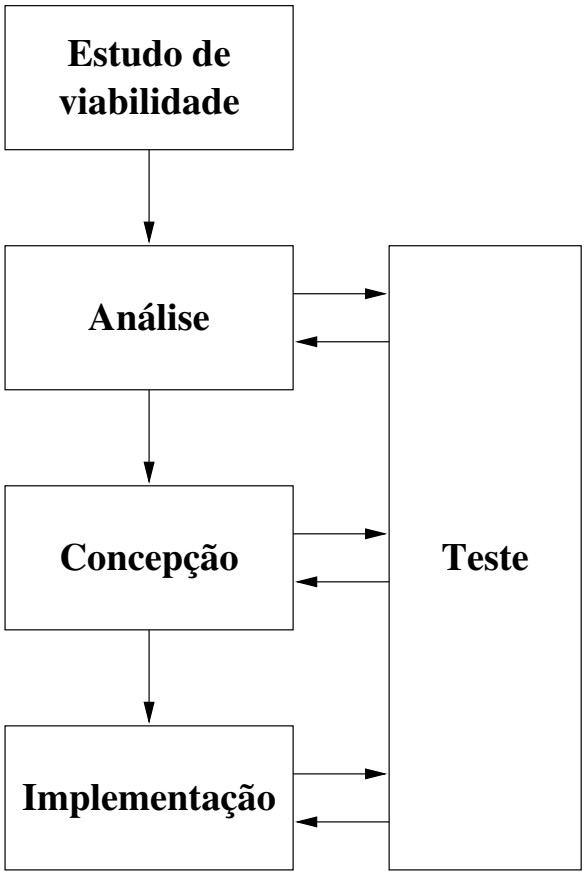


Figura 4.1: O modelo do processo usado na metodologia Midas.

Esta visão está em consonância com a divisão do processo, segundo dois níveis bem distintos: macro e micro [Booch, 1996, pág. 45–6]. O *macro-processo* segue, usualmente, o modelo em cascata, tem associada uma escala temporal maior (meses ou anos) e serve como uma referência de controlo para o *micro-processo*. Este segue, normalmente, o modelo em espiral, tem associada uma escala temporal mais reduzida (semanas, dias ou mesmo horas) e facilita uma abordagem iterativa e incremental no desenvolvimento de sistemas. Garantir o cumprimento do macro-processo é uma tarefa que incumbe aos gestores do projecto, ao passo que o micro-processo é da responsabilidade dum conjunto de projectistas (no limite com cardinalidade igual a um), a quem uma determinada tarefa de desenvolvimento foi atribuída.

Esta perspectiva do processo a dois níveis permite reconciliar duas forças antagónicas. Por um lado, há um certo conflito entre projectos muito cerimoniosos e projectos sem qualquer *cerimónia*. Os primeiros coarctam qualquer assomo de criatividade, pois tudo tem de seguir, cega e fielmente, o estipulado pelo método (exemplo, SSADM [Eva, 1992]), ao passo que os segundos convergem naturalmente para situações caóticas e insustentáveis (exemplo, método

ad-hoc). Por outro, há também horizontes temporais diferenciados, como atrás já se indicou. Assim, ao perspectivar-se o processo a dois níveis distintos, está a encontrar-se uma solução de compromisso que permite conciliar as duas forças de pólos contrários.

A chave para desenvolver sistemas de qualidade reside fundamentalmente em duas vertentes: conseguir um diálogo frutuoso com os especialistas da aplicação e captar correctamente as necessidades do cliente. Daí que, neste trabalho, a fase de análise tenha sido dividida em duas grandes actividades. Na primeira estuda-se a interacção do sistema com o ambiente que o rodeia, o que faz com que o uso de diagramas de contexto e de casos de uso se torne essencial. A segunda actividade *vai dentro* do sistema e identifica os conceitos fundamentais que devem ser representados em termos tanto estruturais como dinâmicos. Esses conceitos são captados por objectos, classes e state-charts, uma vez que se está a considerar uma decomposição orientada ao objecto.

4.3 Fase de análise

A fig. 4.2 enumera, por ordem, os vários passos que devem ser seguidos na fase de análise e indica a informação (predominantemente, sob a forma de diagramas) que é usada como entrada e gerada como saída, em cada um desses passos. Repare-se que cada passo a executar corresponde à criação dum único tipo de informação útil para a persecução da fase de análise. Assume-se que, para a realização dos vários passos, são necessárias, para além dos documentos que são explicitamente referidos na figura, outras informações, por exemplo, entrevistas com utilizadores ou conhecimentos que a equipa de projecto adquiriu por leitura ou consulta de documentação já existente. Embora, nos cap. 5 e 6, se proceda a uma explicação mais elaborada de todos os passos a seguir durante a fase de análise, no âmbito da metodologia MIDAS, faz-se, de seguida, um breve resumo descritivo desses passos.

1. **Captura do ambiente:** Obtém-se o diagrama de contexto que mostra o sistema, entendido como uma entidade única, e todos os actores, vistos como entidades exteriores ao sistema, que com ele interagem.
2. **Modelação funcional:** Obtido o diagrama de contexto, o passo seguinte reside na construção do diagrama de casos de uso, que é entendido como um refinamento daquele. Para cada actor do sistema, especificado no diagrama de contexto, devem identificar-se os respectivos casos de uso a que ele recorre.
3. **Descrição:** Neste passo, escreve-se para cada um dos casos de uso indicados no respectivo diagrama, uma descrição textual que caracteriza os seus aspectos funcionais. Essa descrição deve indicar o tipo de informação utilizada pelo caso de uso.
4. **Transformação:** Os casos de uso são transformados em objectos seguindo um conjunto de regras que foram definidas no âmbito da metodologia. Este passo constitui uma das inovações que a metodologia MIDAS introduziu, pois não é conhecida qualquer outra metodologia que siga uma abordagem que assenta nos mesmos pressupostos.
5. **Modelação do controlado:** Para sistemas com uma forte componente de controlo, este passo é considerado como crucial, pois permite determinar as características do sistema controlado (meios mecânicos, eléctricos, pneumáticos, etc.).
6. **Modelação comportamental:** Tendo em conta o sistema controlado, este passo especifica a forma como o mesmo será usado. A forma de especificar essa utilização não está fixada, dependendo fortemente do sistema em desenvolvimento.

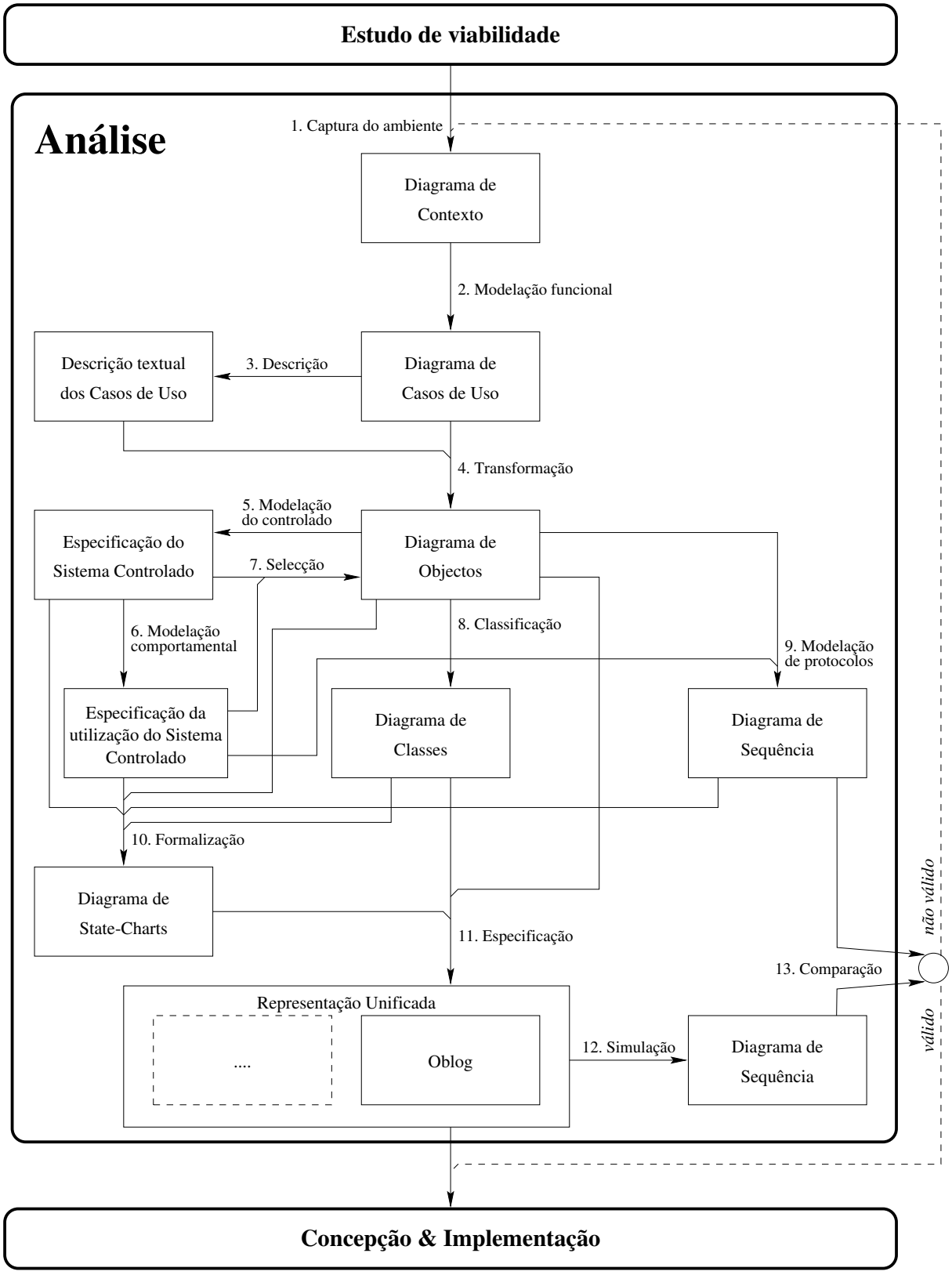


Figura 4.2: Os passos da fase de análise em MIDAS.

7. Selecção: Este passo permite escolher os objectos a ter em consideração para desenvolvimento, de acordo com algum critério. Desta forma, pode restringir-se o sistema àqueles objectos que, por alguma razão, se estimam como importantes.

8. Classificação: Para os objectos criados, indica-se a respectiva classe. Através deste processo de classificação, devem identificar-se objectos que pertençam à mesma classe e também relacionar as várias classes entre si (essencialmente por relações hierárquicas).

9. Modelação de protocolos: Tendo em conta os objectos que constituem o sistema e as ligações entre eles, descrevem-se algumas interacções (aquelas consideradas mais típicas ou relevantes) entre alguns dos objectos do sistema. O número e o nível de pormenor das sequências dependem do sistema em causa e do tipo de informação que consta do documento de requisitos.

10. Formalização: Por análise dos diagramas de sequência e do documento que especifica a utilização do sistema controlado, serão criados diagramas de state-charts, para qualquer objecto/classe cujo comportamento dinâmico se considere merecedor de ser especificado com tal formalismo.

11. Especificação: Neste passo, com base nos diagramas de objectos, de classes e de state-charts, procede-se à criação duma especificação, que descreve essas 3 vistas do sistema. Esta transformação constitui igualmente um dos contributos inovadores da metodologia MIDAS, nomeadamente no que respeita à utilização da linguagem OBLOG como representação unificada de sistemas embebidos.

12. Simulação: Usando a especificação OBLOG como entrada, este passo permite simular o sistema (ou partes dele), sendo criados, como saída, diagramas de sequência que descrevem situações de funcionamento do modelo especificado.

13. Comparação: Com base nos 2 conjuntos de diagramas de sequência obtidos nos passos 10 e 12, faz-se uma comparação dos resultados de ambos os conjuntos, de forma a verificar se o comportamento que se pretende obter (descrito pelos diagramas de sequência do passo 10) corresponde ao comportamento especificado (descrito pelos diagramas de sequência do passo 12).

Note-se que para determinados sistemas, alguns destes passos podem não ser realizados, ou seja, podem ser considerados como dispensáveis. A ordem que se sugere pode igualmente ser alterada.

A ordem dos passos 1 e 2 poderia, eventualmente, ser alterada. Dessa forma, construir-se-ia primeiramente o diagrama de casos de uso e depois, por filtragem, seria obtido o diagrama de contexto. Apesar de se tratar duma possibilidade perfeitamente válida, a ordem escolhida na figura parece ser conceptualmente a mais adequada.

O passo 7, pelo facto de permitir desenvolver por iterações (níveis de prioridade) os objectos que constituem o sistema, enquadra a utilização duma abordagem orientada ao risco no contexto da metodologia MIDAS. É também possível dividir o sistema de forma a criar vários subsistemas (mini-desenvolvimentos) que podem ser atacados por diferentes equipas de projecto. Este passo mostra-se igualmente importante em situações de reengenharia, pois permite "congelar" os objectos que não serão alterados e concentrar os esforços nos objectos que têm de ser novamente desenvolvidos.

A sequência de aplicação dos passos 8 a 10 sugerida não é, de forma alguma, fixa. É perfeitamente válido, depois de obter o diagrama de objectos, alterar essa ordem. Por exemplo, pode ser adequado, em determinadas situações, criar primeiramente os diagramas de sequência, depois os diagramas de state-charts e finalmente o diagrama de classes.

O passo 9 pode também ser aplicado ao diagrama de contexto, criando-se assim diagramas de sequência para interacções entre o sistema e os seus actores.

O passo 11 impõe a geração duma especificação unificada usando uma dada linguagem. Neste trabalho, foi seleccionada a linguagem Oblog e o cap. 6 aborda, duma forma exaustiva, a transformação dos diagramas UML (classes, objectos e state-charts) em código Oblog. Todavia, esta linguagem não é a única que pode ser usada neste passo e, como a fig. 4.1 ilustra, a metodologia Midas deixa em aberto a possibilidade de utilizar-se uma outra linguagem como representação intermédia.

Dado que, no passo 12, é possível simular o funcionamento do sistema, mais concretamente o repositório Oblog obtido, pode afirmar-se que a metodologia Midas facilita a possibilidade de seguir, durante o processo de desenvolvimento, uma abordagem operacional.

O resultado da comparação que se obtém no passo 13 permite que se possa determinar se o sistema está ou não correctamente especificado. Em caso afirmativo, o repositório Oblog é usado como entrada para a fase de concepção. Caso contrário, deve executar-se uma nova iteração de análise na expectativa de encontrar e corrigir totalmente as não-conformidades detectadas na comparação. Claramente, esta possibilidade enquadra o uso dum modelo de processo iterativo (nomeadamente, o modelo em espiral).

A repetição do ciclo de análise pode realizar-se por outros motivos que não apenas a detecção de diferenças entre os dois conjuntos de diagramas de sequências. Por exemplo, pode ser conveniente, numa primeira iteração, não pormenorizar alguns objectos, deixando a sua descrição o mais abstracta possível. Em iterações ulteriores, far-se-á então a pormenorização desses objectos, o que poderá implicar a criação de novos (sub-)objectos, novas classes e consequente reestruturação da hierarquia de classes e a especificação de novos state-charts.

4.4 Fases de concepção e implementação

Como já se referiu, o maior contributo desta tese, para a definição da metodologia Midas, verificou-se na fase de análise, tendo outros trabalhos, no âmbito do projecto referido na secção 1.5, focado nas questões técnicas e metodológicas relacionadas com as fases de concepção e de implementação. Todavia, far-se-á, nesta secção, um pequeno levantamento sobre estas questões, de modo a enquadrar o trabalho desta tese num âmbito mais lato.

Durante o levantamento de requisitos, as restrições de tempo real devem ser recolhidas e analisadas, o que se faz associando essas restrições aos casos de uso do sistema. O desenho do diagrama de objectos não é, regra geral, afectado por essas restrições pois trata-se de um modelo independente da plataforma de implementação. Contudo, pode fazer-se associar as restrições de tempo-real de cada caso de uso aos objectos a que esse caso de uso deu origem, facilitando assim o desenvolvimento do sistema nas fases de concepção e implementação. É apenas nestas fases que essas restrições devem ser consideradas.

Na fase de concepção, o diagrama de objectos é a base para introduzir, nos modelos, os pormenores de implementação a tomar em consideração. Assim, para que essa transformação seja possível, devem executar-se os seguintes passos:

- Identificar o ambiente de implementação que vai suportar o sistema.
- Desenvolver uma primeira versão do ambiente de concepção.
- Descrever a forma como os objectos interactuam.

O processo de concepção pode ser conduzido segundo duas alternativas: tradução ou elaboração. Numa abordagem tradutora, o modelo de análise é transformado, duma forma mais ou menos automática, num sistema executável, através dum tradutor. É importante colocar grande cuidado na construção do tradutor, que é normalmente usado num dado domínio de aplicação. A outra abordagem elabora o modelo de análise, através da adição de pormenores de concepção, até o sistema estar completamente especificado. O uso do utilitário OBLOG impõe um processo de concepção baseado em tradução, razão pela qual essa abordagem foi adoptada neste trabalho.

A linguagem (gráfica e textual) OBLOG é usada, neste trabalho, como base para especificar os sistemas embebidos que se pretendem desenvolver com a metodologia MIDAS. Neste sentido, a linguagem OBLOG é vista como uma representação unificada, por permitir especificar partes que tanto podem ser implementadas em software como em hardware.

A partir da especificação em OBLOG pode gerar-se código em C/C++ e em VHDL, para as partes a implementar em software e em hardware, respectivamente. O cap. 6 aborda a criação do repositório OBLOG, a partir dos diagramas UML gerados nas tarefas anteriores.

A possibilidade de gerar automaticamente código C/C++, a partir das especificações OBLOG, torna estas executáveis, uma vez que pode obter-se um programa passível de ser executado para efeitos de simulação funcional. Adicionalmente, por não ser obrigatória a utilização de OBLOG como representação unificada, a escolha acertada de outras linguagens para esse fim (por exemplo, JAVA) permite igualmente que se possam executar as especificações.

Uma questão que terá igualmente de ser tratada na fase de concepção é a partição hardware/software das funcionalidades do sistema. Dependendo do nível de granulosidade (i.e. as unidades mínimas), a partição, segundo um determinado critério, poderá determinar a reestruturação dos componentes do sistema (no caso da metodologia MIDAS, os objectos). Para partir um sistema pelos vários componentes da plataforma alvo seleccionada, pode ser necessário dividir um objecto em vários outros, agrupar dois ou mais objectos num só ou ainda recorrer a outras hipóteses não tão directas (exemplo, construir um novo objecto com base em partes de outros).

4.5 Contributos

A metodologia MIDAS, durante a sua elaboração, recebeu contributos provenientes de inúmeras fontes, os quais tiveram uma influência decisiva no resultado final obtido (fig. 4.3).

De seguida, far-se-á uma explicação da fig. 4.3, indicando a área coberta por cada uma das contribuições. Começando pela parte inferior da figura, UML foi, como já se referiu, a notação escolhida por ser de ampla utilização, mas sobretudo, por ter todos os mecanismos necessários à modelação de sistemas embebidos de elevada complexidade. Apesar de UML ter sido inicialmente criada para sistemas software, nos últimos tempos, têm surgido diversas propostas que integram a notação UML no âmbito do desenvolvimento de sistemas embebidos e de tempo-real [Douglass, 1998] [McLaughin e Moore, 1998] [Lanusse et al., 1998] [Lyons, 1998] [Selic e Rumbaugh, 1998]. Dos vários diagramas UML existentes, aqueles que foram considerados como mais relevantes para o desenvolvimento de sistemas embebidos, no contexto da metodologia MIDAS, são os seguintes:

- **Diagramas de casos de uso:** são usados para captar a perspectiva funcional do sistema, na óptica dos seus utilizadores.

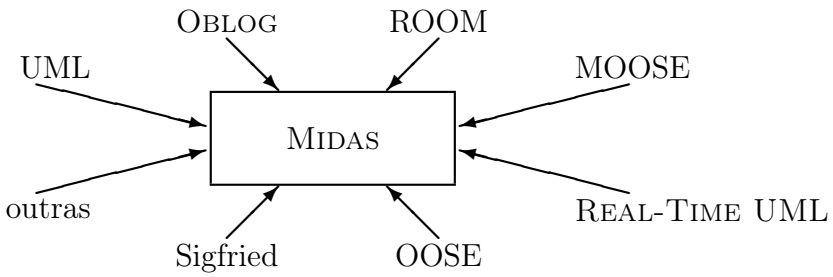


Figura 4.3: As contribuições mais importantes para a definição da metodologia MIDAS (fase de análise).

- **Diagramas de objectos:** mostram a configuração estática do sistema e as relações entre os objectos que constituem o sistema.
- **Diagramas de classes:** armazenam as especificações de componentes prontos a usar na construção de sistemas e explicitam as relações de herança entre esses componentes.
- **Diagramas de state-charts:** são utilizados para definir o comportamento dinâmico de algumas classes.
- **Diagramas de sequência:** permitem documentar um conjunto de interacções típicas entre objectos e/ou actores do sistema.

O utilitário OBLOG serve de sustentáculo à metodologia e tem associada uma linguagem de modelação orientada ao objecto (também designada OBLOG). Esta ferramenta tem uma característica extremamente interessante que consiste num mecanismo de geração automática de código a partir das especificações (gráficas e/ou textuais) do sistema. A vantagem deste mecanismo resulta da possibilidade de serem desenvolvidos novos geradores (*scripts*) para linguagens ainda não consideradas ou para linguagens já consideradas, mas segundo uma abordagem distinta[5], o que torna o utilitário completamente aberto, facilitando assim a sua integração noutros sistemas já existentes. Relativamente à linguagem, houve a necessidade de adaptá-la às características próprias dos sistemas embebidos, através da selecção dum sub-conjunto apropriado, pois ela foi inicialmente concebida para sistemas puramente software.

As metodologias ROOM [Selic et al., 1994], MOOSE [Morris et al., 1996] e REAL-TIME UML [Douglass, 1997, Douglass, 1998], por tratarem o desenvolvimento de sistemas hardware/software segundo o paradigma dos objectos, serviram como importantes referências para a definição de toda a metodologia MIDAS.

Da metodologia OOSE [Jacobson et al., 1992], adoptaram-se os casos de uso e algumas das recomendações que permitem, a partir daqueles, obter os diversos objectos que compõem o sistema em desenvolvimento.

Sigfried teve, no trabalho, uma influência decisiva na forma como algumas práticas do paradigma orientado ao objecto são interpretadas, nomeadamente questões relacionadas com os mecanismos que devem estar disponíveis nos diagramas de objectos e de classes [Sigfried, 1996].

Apesar das referências anteriores constituírem o núcleo duro das contribuições para a metodologia MIDAS, é evidente que outras contribuições também forneceram mais-valias, por mais pe-

[5]É fácil imaginar a necessidade de gerar código VHDL para, por exemplo, documentação, simulação ou síntese.

quenas que sejam, ao resultado final obtido. Não sendo possível identificá-las a todas elas ou até não havendo a percepção de quais elas são, ficam assim enquadradas num único tópico.

4.6 Resumo final

Neste capítulo, justificou-se a necessidade duma metodologia orientada ao objecto para desenvolvimento de sistemas embebidos, apresentando primeiramente a utilidade duma metodologia e depois as vantagens que decorrem se essa metodologia seguir o paradigma dos objectos.

Neste pressuposto, foi apresentada a metodologia MIDAS, que constitui o principal contributo desta tese. Os elementos que foram tidos por fundamentais na definição da metodologia MIDAS foram o uso da abordagem operacional, de especificações gráficas e multi-vista, e de modelos orientados ao objecto. Foi ainda descrito o modelo de processo que a metodologia preconiza.

De seguida, foram levantadas algumas questões relacionadas com as fases de análise, concepção e implementação, de modo a enquadrar a utilização da metodologia num âmbito mais alargado. Foi dado especial relevo à fase de análise, por ser aí que se centram os objectivos desta tese.

A finalizar o capítulo, foram enumerados os principais contributos que influenciaram a elaboração da metodologia MIDAS, nomeadamente, UML, OBLOG, ROOM, MOOSE, REAL-TIME UML e OOSE.

Capítulo 5

A Análise na Metodologia MIDAS

Melhor é alguma coisa que nada.

Sumário

Neste capítulo, indicam-se quais as notações e os métodos para as diversas etapas da fase de análise. A apresentação é feita tomando como critério de organização os diversos diagramas que devem construir-se ao longo da fase de análise: contexto, casos de uso, objectos, classes, estados e sequência. É feita também uma apresentação sobre as questões relacionadas com a comunicação entre objectos, relevante para os todos os diagramas em que existam ligações inter-objectos. Relativamente aos diagramas de objectos, apresentam-se as categorias de objectos (interface, entidade e função) relevantes para descrever um sistema, indica-se a forma de compor, repetir, criar e eliminar objectos e refere-se a forma como questões temporais devem ser introduzidas nos modelos. Em relação aos diagramas de classes, referem-se questões relativas à composição, à herança e em que circunstâncias as classes devem ser concretas ou abstractas. Quanto aos diagramas de state-charts, abordam-se as regras para a sua utilização, o modelo temporal (síncrono ou assíncrono) que lhes está associado e a forma como a sua utilização se repercute na herança entre classes.

Índice

Nas secções seguintes, pretende, no âmbito da metodologia MIDAS, indicar-se como a fase de análise é seguida, indicando o meta-modelo usado, bem como os métodos a seguir para modelação de sistemas embebidos. A divisão foi feita de acordo com os documentos a criar, como a fig. 4.1 mostra; contudo não há uma relação unívoca entre os documentos e as secções pois, em alguns casos, incluíram-se vários documentos numa só secção. A criação do repositório OBLOG (passo 11) foi remetida para o cap. 6, devido à sua grande extensão e por ser uma das tarefas que constitui um dos contributos desta tese. Os métodos a adoptar são constituídos por algumas recomendações que se julgam apropriadas para criar os modelos. Contudo, essas recomendações são um complemento à experiência e à criatividade do projectista e não pretendem, de modo algum, substituí-las.

5.1 Diagramas de contexto

Qualquer sistema, para ter algum tipo de utilidade, tem obrigatoriamente de estabelecer ligações ao seu exterior. Um sistema completamente isolado do resto do mundo, sem quaisquer ligações externas (entradas e saídas), mesmo estando activo, apresenta-se, dum ponto de vista prático, completamente inútil, pois não é possível actuar sobre o sistema, nem observar os resultados da sua actividade. Nesta linha de raciocínio, qualquer sistema implica necessariamente um ambiente composto por entidades que interagem com esse sistema.

O primeiro passo no desenvolvimento dum sistema consiste, pois, na definição do diagrama de contexto [Morris et al., 1996, pág. 70] [Douglass, 1998, pág. 52] que tem o objectivo de definir, com clareza, a fronteira entre o sistema e o seu ambiente, o que se mostra útil para delimitar o contorno do sistema a desenvolver. Demarcar a fronteira do sistema significa identificar aquilo que está dentro do sistema (e que, portanto, tem de ser desenvolvido) e aquilo que está fora (e que não deve ser desenvolvido, mas que deve ser tido em conta pois interage com o sistema) [Schneider e Winters, 1998, pág. 11].

O diagrama de contexto mostra o sistema, representado como uma entidade única, rodeado por todos os actores que com ele interactuam. Além disso, permite caracterizar o interface do sistema com o exterior, ou seja, as mensagens e os eventos que fluem entre o sistema e o respectivo ambiente. Todos os actores do sistema são-lhe externos, pelo que não devem ser desenvolvidos no âmbito do projecto do sistema. Desta forma, há uma definição precisa daquilo que é o sistema que interessa desenvolver.

Apesar da notação UML não suportar, explícita e directamente, a descrição de diagramas de contexto, é possível adaptar os diagramas de caso de uso e de objectos, de modo a usá-los para esse propósito. Assim, usam-se actores, como os usados nos diagramas de caso de uso (subsecção 3.3.1), para especificar as entidades externas que comunicam com o sistema e um objecto para representar todo o sistema. Pode também olhar-se o diagrama de contexto, numa perspectiva alternativa, como sendo o diagrama de casos de uso de mais alto nível, que é constituído por um único caso de uso, representativo de toda a funcionalidade disponibilizada pelo sistema.

A estratégia para criar o diagrama de contexto consiste nos 3 passos seguintes:

1. Construir uma lista de actores, entradas e saídas.
2. Desenhar o diagrama de contexto, escolhendo o tipo adequado para cada uma das ligações entre actores e o sistema.
3. Especificar pormenorizadamente as ligações numa tabela.

Identificar os actores dum sistema, apesar de parecer uma tarefa simples, pode revelar-se difícil e ingrato, pois muitas vezes é complicado justificar se uma dada funcionalidade está dentro ou fora do sistema. Normalmente, é sempre necessária alguma iteração nesse processo de identificação dos actores, até se atingir uma solução estável. Algumas recomendações a atender, para proceder à identificação desses objectos (externos ao sistema) são as seguintes [Douglass, 1998, pág. 54–5]:

1. O enunciado do problema refere-se explicitamente ao objecto, como, por exemplo, quando o sistema tem de interactuar com dispositivos.
2. O objecto já está disponível no ambiente do domínio de aplicação, ou seja, não será desenvolvido como parte do sistema.
3. O objecto envia ou recebe informação do sistema, mas não é incluído no sistema.
4. O objecto é uma pessoa que monitoriza as saídas do sistema, introduz dados, requisita a execução de comandos, ou é directamente afectada pelas acções do sistema.

Quando nenhuma destas hipóteses se aplica a um dado objecto e ele é importante no contexto do problema, então esse objecto deve ser desenvolvido no âmbito do sistema em causa e nele incorporado.

Outra forma alternativa ou complementar de encontrar os actores do sistema consiste em colectar as respostas a uma série de questões que a equipa de projecto pode colocar a si mesma [Schneider e Winters, 1998, pág. 12]:

1. Quem usa o sistema?
2. Quem instala o sistema?
3. Quem arranca o sistema?
4. Quem mantém o sistema?
5. Quem desliga o sistema?
6. Que outros sistemas usam o sistema?
7. Quem recebe informação do sistema?
8. Quem disponibiliza informação ao sistema?
9. Algo sucede automaticamente no momento de arranque?

Depois de identificados os actores do sistema, deve fazer-se uma descrição sucinta do papel que cada um deles desempenha relativamente ao sistema. Uma vez que se assume um processo de desenvolvimento iterativo, à medida que forem obtidos novos conhecimentos sobre os actores, em qualquer fase do projecto, deve fazer reflectir-se esses conhecimentos na documentação do projecto (nomeadamente no diagrama de contexto e nas descrições dos papéis dos actores), de forma a manter actualizada toda a informação.

Apesar de merecer normalmente pouca atenção, a escolha de nomes apropriados para os casos de uso revela-se importante pois aumenta a legibilidade do respectivo diagrama, motivo pelo qual se recomenda, em Midas, que seja dispendido algum tempo na atribuição de nomes idóneos.

Esta recomendação é igualmente válida para qualquer outro tipo de diagrama. Nas classes e nos objectos, além do nome destes, deve também ser prestada especial atenção aos atributos e operações. Nos state-charts, os nomes dos estados devem também ser merecedores de algum cuidado.

5.2 Comunicação inter-objectos

Apesar desta secção estar imediatamente após a secção dedicada aos diagramas de contexto, as interligações de comunicação aqui descritas são igualmente aplicáveis a todos os diagramas em que existam ligações entre objectos, nomeadamente, diagramas de classes, diagramas de objectos, diagramas de sequência e diagramas de colaboração.

Regra geral, as metodologias para desenvolvimento de software usam apenas, como meio de comunicação, mensagens, situação que se repete em algumas metodologias para sistemas hardware/software (ROOM, por exemplo). A mensagem é um mecanismo de comunicação ponto a ponto (objecto a objecto), pelo que não é possível a difusão de mensagens. A representação completa do comportamento dum sistema que contém partes em software e em hardware, à luz dum meta-modelo que inclui unicamente o mecanismo de passagem de mensagens para comunicação entre objectos, apesar de ser possível, mostra-se uma solução demasiado vinculada à forma como os componentes de software comunicam entre si [Morris et al., 1996, pág. 66]. Assim, um meta-modelo baseado apenas em mensagens limita significativamente uma característica desejada para os modelos, que consiste no não comprometimento destes com a solução final (seja em hardware, seja em software).

Alguns exemplos concretos talvez permitam elucidar melhor este assunto. Entre os objectos candidatos para implementação em hardware encontram-se aqueles que lidam com sinais de elevada largura de banda, presentes, por exemplo, em algumas aplicações de visão e de áudio. Modelar as ligações a esses objectos através de protocolos usando unicamente mensagens, além de ser bastante inadequado, poderia, nessa situação, confundir a equipa de projecto sobre qual a melhor solução de implementação para esses objectos.

Também há situações em que a ocorrência de eventos é relevante para vários componentes, pelo que, nesses casos, convém, por questões de expansibilidade, que o objecto, responsável por assinalar o evento, não necessite de conhecer a quem esse evento é relevante. Tal situação não é facilmente modelada com mensagens, pois, recorrendo unicamente a esse mecanismo, será inevitável serem explicitamente enviadas mensagens a todos os objectos sensíveis a esse evento. O objecto responsável pela sinalização do evento terá, portanto, a ingrata tarefa de conhecer a que objectos e quando deve dar conhecimento da ocorrência do evento.

Finalmente, certos objectos podem ser responsáveis por disponibilizar continuamente certos valores, ao longo do tempo, sem necessidade desses objectos conhecerem quem usa, e quando, essa informação. Como exemplos típicos, considere-se um relógio que disponibiliza continuamente a hora do dia, um termómetro que proporciona a temperatura, ou um sensor que fornece em contínuo o valor do nível de concentração de gás num dado ambiente fechado (quarto). Embora seja sempre possível disponibilizar a informação mediante o estabelecimento dum protocolo envolvendo mensagens (o objecto que necessita da informação envia um pedido ao objecto que a possui, que imediatamente responde ao pedido com o valor actualizado da informação pela qual é responsável), tal solução não é a mais natural, podendo até ser entendida como forçada.

Em sistemas reactivos, como aqueles que este trabalho contempla, o tratamento dos eventos e da informação depende tipicamente do estado do sistema, que se encontra distribuído pelos objectos que o compõem. Este facto requer que cada objecto tenha acesso aos eventos e às informações que lhe são relevantes, de acordo com as necessidades e o estado desse objecto. Muitas vezes, o controlo dinâmico e a transformação da informação são os aspectos mais importantes de concepção, sendo irrelevante o controlo sobre o acesso à informação. Assim, o apertado controlo sobre o acesso que é imposto pela comunicação por mensagens apresenta-se

demasiado restritivo para sistemas embebidos, motivo pelo qual são propostos alguns novos tipos de ligações inter-objectos.

Em Midas, são possíveis 4 tipos distintos de interligações entre objectos:

- **Interacção**: Retrata o habitual mecanismo de passagem de mensagens em software, sendo a sua semântica similar à chamada duma função. Trata-se dum mecanismo síncrono de comunicação entre 2 objectos, em que o emissor requisita uma dada operação ao receptor, ficando aquele em suspenso à espera que este responda, através do envio dum resultado.
- **Evento**: É usado, por um objecto, para sinalizar as mudanças do seu próprio estado que possam ser relevantes a outros objectos. Não há neste mecanismo a noção de sincronismo, dado que a natureza e o tempo de resposta não são controlados pelo objecto emissor. Trata-se dum mecanismo de comunicação fundamental para sistemas reactivos, usado com frequência na especificação dos diagramas de estados dos objectos.
- **Fluxo discreto de informação**: Exprime a transferência de dados entre objectos, duma forma discreta. Neste mecanismo não existe a noção de sincronismo, dado que o receptor da informação pode ler qualquer valor anteriormente gerado sempre que dele necessitar. Não há, portanto, qualquer tipo de controlo por parte do emissor sobre quando essa leitura é feita, pois aquele apenas controla o valor da informação, já que é o responsável pela sua actualização.
- **Fluxo contínuo de informação**: Este mecanismo de ligação é muito idêntico ao anterior, mas pressupõe a transferência de dados entre objectos, dum modo contínuo. Directamente, o receptor da informação tem apenas disponível para uso o último valor de informação (i.e. o valor actual).

Os modos assíncronos de comunicação implicam, na prática, que o objecto responsável pela informação, após a disponibilizar, continua a sua execução, independentemente da forma como essa informação é tratada pelo receptor. Em comunicações síncronas, sucede o contrário, ou seja, o emissor da informação fica à espera do tratamento dado pelo receptor para poder prosseguir a sua execução.

O modo assíncrono de comunicação permite ao emissor continuar a executar logo após a mensagem ter sido despachada. A principal vantagem associada a este modo reside no facto de o período de tempo, durante o qual o objecto emissor está a tratar da comunicação, ser curto. No modo síncrono de comunicação, o emissor fica bloqueado até receber uma resposta à mensagem inicial. Durante esse período, não pode responder a outros estímulos que, eventualmente, lhe sejam enviados. A vantagem desta forma de comunicação consiste no nível de controlo relativamente à ordem das actividades que é implicitamente disponibilizado.

No modo assíncrono, não é devolvido qualquer resultado ao emissor, o que não significa que uma outra mensagem não possa ser enviada ao emissor da mensagem inicial. Não há, neste modo, é um relacionamento tão evidente e explícito entre a mensagem e a resposta. O modo síncrono apresenta alguns problemas: bloqueios, menos tempo para responder a mensagens, dependência do emissor de mensagens relativamente aos receptores.

Foram já indicadas algumas situações e dados alguns exemplos, em que é relevante a utilização de fluxos contínuos de informação. Uma das limitações deste tipo de ligação prende-se com o facto de só ser disponibilizado, ao receptor, o valor actual da informação. Todos os valores anteriores não estão directamente disponíveis no fluxo, i.e. perderam-se, a menos que o receptor os tenha armazenado localmente. Por este motivo, a utilização de fluxos discretos de informação, por obviar esta limitação, revela-se importante, sempre que for necessário disponibilizar ao

receptor uma sequência de valores relativos a uma dada informação.

Um exemplo típico da necessidade de usar fluxos discretos de informação ocorre quando se tem de criar um historial (*log*), ou seja, sempre que se pretende guardar valores, dum modo periódico, para posterior manipulação (por exemplo, para pesquisa de valores fora dum dado intervalo).

No âmbito da metodologia MIDAS, propõem-se duas formas alternativas de indicar qual o tipo duma determinada interligação, usando o mecanismo de estereótipos de UML. Assim, os 4 tipos de interligações podem representar-se da forma que a fig. 5.1 ilustra, não estando a sua aplicação limitada ao diagrama de contexto; pode também usar-se noutros diagramas em que haja comunicação entre objectos ou entre objectos e actores (por exemplo, nos diagramas de objectos e nos diagramas de interacção).

Tipo de interligação	**Símbolo gráfico**	**Estereótipo**
Interacção		≪interaction≫
Evento		≪event≫
Fluxo discreto de informação		≪discrete info≫
Fluxo contínuo de informação		≪continuous info≫

Figura 5.1: Estereótipos propostos para descrever os tipos das interligações.

A fig. 5.2 mostra um exemplo de cada uma dessas interligações. O objecto objA, que fica no lado oposto ao da seta, é o cliente da interacção, ou seja, é responsável por iniciar a comunicação, enquanto o objecto objB, mais próximo da seta, desempenha, nesse caso, o papel de servidor. O objecto objC é o emissor do evento, devendo activá-lo sempre que o seu estado o determinar. O eventual tratamento desse evento fica a cargo do objecto objD. Para os fluxos de informação, os objectos objE e objG são responsáveis por manter aqueles actualizados, podendo os objectos objF e objH ler os conteúdos informativos sempre que o desejem.

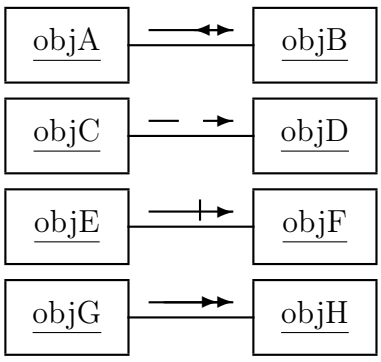


Figura 5.2: Objectos ligados pelos 4 tipos de interligações disponíveis em MIDAS.

Para concluir esta discussão sobre formas de interligação entre objectos, refira-se que é ainda possível considerar mais duas hipóteses. A primeira, a que se dá o nome de *grupo*, refere-se a um mecanismo de abstracção que permite concentrar, num só elemento gráfico, várias interligações de mais baixo nível. Trata-se duma forma muito útil de esconder pormenores que a um dado nível de abstracção podem não ser relevantes, também disponível na metodologia MOOSE, sob o nome de *bundle*. Por exemplo, se um utilizador tiver disponíveis vários comandos para actuar num dado sistema, pode não interessar, numa primeira abordagem, ter de indicar todos eles. A solução, nesse caso, passa, como a fig. 5.3 mostra, pela introdução dum grupo que os representa a todos, ficando para altura posterior a especificação de quais os comandos que constituem esse grupo.

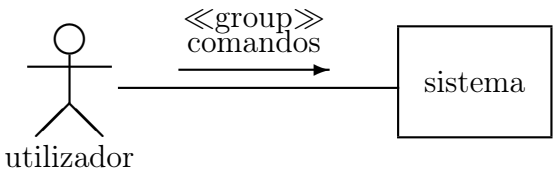


Figura 5.3: Um grupo de interligações.

Se um grupo for constituído por interligações do mesmo tipo, está-se perante um grupo homogéneo, podendo assim adoptar-se o mesmo símbolo gráfico. Para distinguir um grupo homogéneo duma interligação simples, é necessário, relativamente ao primeiro, acrescentar um estereótipo ≪group≫. Caso o grupo seja composto por interligações de tipos distintos, o grupo diz-se heterogéneo, devendo, desta vez, adoptar-se um símbolo diferente. Nesse sentido, sugere-se o uso duma seta simples (⟶) e do estereótipo ≪group≫, para esta situação.

O uso do estereótipo na última situação pode parecer desnecessário, mas tal não corresponde à verdade, pois é ainda possível utilizar a seta simples para representar interligações simples, mas cujo tipo não é conhecido na altura. Trata-se, portanto, dum elemento gráfico que não fará parte da documentação final dum projecto, mas que, em termos práticos, será útil considerar, dada a grande probabilidade de, no decurso dum projecto, se desconhecer o tipo exacto duma interligação. Assim, o uso da seta simples sem a presença do estereótipo ≪group≫ representa uma interligação entre dois objectos que foi identificada, mas cujo tipo ainda não foi possível determinar.

5.3 Diagramas de casos de uso

Apesar dos casos de uso serem usados em processos de desenvolvimento orientados ao objecto, não há nenhuma característica intrínseca aos casos de uso que possa ser considerada como puramente "orientada ao objecto". Assim, os casos de uso podem também ser usados em desenvolvimentos que não seguem o paradigma dos objectos. Contudo, há um amplo reconhecimento que os casos de uso são uma técnica que se adapta muito bem à abordagem orientada ao objecto, nomeadamente para descobrir e, depois, especificar o comportamento dum sistema, durante a fase de análise, motivo pelo qual se optou pela sua adopção no âmbito da metodologia Midas.

O conjunto de todos os casos de uso dum sistema representa a funcionalidade completa do sistema segundo a perspectiva dos seus utilizadores. Para identificar os casos de uso dum sistema, a melhor estratégia (ou, pelo menos, a estratégia mais simples) consiste em tentar encontrar,

primeiramente, os actores desse sistema e, depois, para cada um daqueles, colectar uma lista de possíveis casos de uso. Cada fluxo completo de eventos, iniciado por um actor, representa um caso de uso do sistema em análise. Os actores do sistema já foram obrigatoriamente identificados no diagrama de contexto, pelo que resta a tarefa de colectar uma lista de casos de uso.

É talvez preferível ver a construção dos diagramas de contexto e de casos de uso como um processo iterativo, filosofia que, aliás, tem sido constantemente defendida neste trabalho. Dito de outra forma, se durante a construção do diagrama de casos de uso forem detectadas algumas incongruências (por exemplo, a necessidade de introduzir um novo actor), as alterações a propor devem também fazer-se reflectir no diagrama de contexto, uma vez que toda a documentação do projecto deve estar sempre actualizada.

Os casos de uso podem ser decompostos em outros casos de uso, repetindo-se o processo até eventualmente se chegar aos cenários que mostram sequências pormenorizadas de interacção entre objectos. Assim, outra possibilidade para construção dos diagramas de casos de uso consiste na identificação, em primeiro lugar, dos cenários do sistema, a partir dos quais é possível, por agregação, inferir os casos de uso.

Os cenários devem ser descritos, usando, para o efeito, diagramas de sequência. Alternativamente, poderiam usar-se diagramas de colaboração, uma vez que estes dois tipos de diagramas são isomorfos, sendo usados em UML para o mesmo propósito [Booch et al., 1999, pág. 25]. É possível criar utilitários que transformam automaticamente, sem perda de informação, um diagrama de sequência num diagrama de colaboração e *vice-versa*, pelo que a escolha, de qual deles usar, parece depender fortemente de motivos pessoais [Bergner et al., 1998]. Assim, na metodologia MIDAS, incentiva-se a utilização de diagramas de sequência em detrimento dos diagramas de colaboração, por quatro razões principais:

1. Os diagramas de colaboração não são tão intuitivos (leia-se de maior legibilidade) como os diagramas de sequência e, além disso, não acrescentam nenhuma informação adicional.
2. Deve evitar-se o uso de diagramas de colaboração, em fases iniciais da análise, uma vez que é susceptível existirem modificações nos objectos que compõem o sistema, o que, a acontecer, iria implicar uma alteração profunda nos diagramas de colaboração[1].
3. As restrições temporais são melhor especificadas nos diagramas de sequência, o que se revela uma característica relevante para o desenvolvimento de sistemas de tempo-real.
4. O utilitário OBLOG permite criar, durante a simulação do sistema, diagramas de sequência, o que se mostra útil para validar essa simulação, caso tenham sido gerados diagramas de sequência em passos anteriores da fase de análise.

Do que atrás foi referido, não se conclua, erradamente, que o uso de diagramas de colaboração não está permitido em MIDAS. O que se defende é que, pelo facto de existirem em UML dois diagramas diferentes para o mesmo efeito, deve optar-se apenas por um, com o intuito de reduzir, ao mínimo indispensável, os elementos de modelação que precisam de ser dominados. Contudo, se o projectista achar oportuna a utilização de diagramas de colaboração, num determinado projecto, deve fazê-lo.

[1] Pode questionar-se se, igualmente, não aconteceriam alterações num diagrama de sequência, caso os objectos do sistema fossem modificados. A resposta é afirmativa, mas há a convicção que, na maioria das situações, as alterações nos diagramas de sequência são menos drásticas. Exemplificando, considere-se que é necessário introduzir uma nova mensagem entre dois objectos: num diagrama de colaboração terão que renumerar-se todas as mensagens seguintes, enquanto que num diagrama de sequência há apenas que colocar essa nova mensagem na sua posição, uma vez que a ordem entre as mensagens é dada pela sua posição relativa.

5.3.1 Discussão

A semântica associada aos casos de uso levanta alguns problemas, cuja solução se pretende dar de seguida. A primeira questão refere-se à possibilidade de haver casos de uso sem qualquer ligação, implícita ou explícita, a actores. Tal hipótese não é permitida em UML e, à primeira impressão, pode parecer que esse caso de uso contraria o propósito dos casos de uso, como mecanismo de modelação, para captar as funcionalidades que o sistema disponibiliza aos seus utentes. No entanto, essa possibilidade parece útil para modelar tarefas que têm de ser executadas periódica e automaticamente e sem qualquer intervenção externa (iniciadas internamente por iniciativa do próprio sistema e não dum actor[2]). Por exemplo, um sistema que tenha que fazer, no final do dia, alguma operação automática, tipo estatísticas, historial (*logs*) ou reciclagem de memória (*garbage collection*), deve ele próprio ser o responsável por iniciar internamente essa operação, podendo eventualmente os resultados desta serem disponibilizados a um actor. Sugere-se a utilização dum estereótipo ≪internal≫ associado aos casos de uso que representam essas tarefas iniciadas internamente.

Em UML, assume-se que se um caso de uso A é estendido por um caso de uso B, então B tem, implicitamente, ligações a todos os actores que estão ligados a A. Na opinião do autor, esta não é uma boa suposição, pois pode, em determinadas situações, haver interesse em associar actores distintos a cada caso de uso. Para ilustrar esta situação, retome-se o exemplo da fig. 3.9(a), em que pode considerar-se pertinente distinguir quem tem direito a "Editar dados confidenciais" (apenas os médicos) e que tem acesso a "Editar dados" (enfermeiros e contínuos). Com a semântica actual de UML, esta hipótese não pode ser modelada. Neste sentido, propõe-se a utilização dum novo estereótipo ≪refines≫, que é uma variante da relação ≪extends≫, mas que não assume a ligação aos actores do caso de uso base.

Admita-se que um caso de uso A é estendido por um caso de uso B. Nesta situação, pode interrogar-se se o caso de uso B tem obrigatoriamente de ter relações ≪uses≫ com os mesmos caso de uso a que A está também ligado por relações ≪uses≫. Em princípio, tal obrigatoriedade parece redundante, pois sendo B uma extensão de A, é natural que B também use os casos de uso a que A está ligado. A recomendação em Midas vai no sentido de apenas incluir a relação ≪uses≫, quando o caso de uso estendido introduzir funcionalidade própria que use algum desses casos de uso e não a incluir em caso contrário [Bergner et al., 1998]

Quando o diagrama de casos de uso se começa a tornar muito complexo (i.e. com muitos casos de uso e actores), há várias soluções para lidar com essa complexidade:

1. Agrupar vários casos de uso, relacionados de alguma forma, num único caso de uso (ou num pacote).
2. Criar vários diagramas de casos de uso, tendo cada um deles uma parte da funcionalidade do sistema.
3. Criar um diagrama de casos de uso, para cada um dos actores (ou para cada um dos casos de uso).

Estas duas últimas soluções envolvem alguma redundância, o que obriga a relacionar os vários diagramas construídos, por forma a encontrar os pontos de sobreposição. Neste sentido, o uso de utilitários CASE pode revelar-se muito vantajoso nesta tarefa.

Uma vez que os casos de uso dividem funcionalmente o sistema, em Midas, permite atribuir-se uma referência numérica a cada caso de uso, como sucede igualmente nos diagramas de

[2]Na pág. 140, apresenta-se a justificação para o tempo ser considerado um componente interno do sistema e não um actor que com ele interage.

fluxo de dados (DFDs). A aposição dessa referência, no caso de uso, é feita através dum valor etiquetado (por exemplo, {ref=9}). Quando um caso de uso é refinado em vários casos de uso menos abstractos, a numeração destes usa, como prefixo, a referência do caso de uso mais abstracto e acrescenta a numeração interna (para o exemplo anterior, se existissem 3 casos de uso, estes teriam as referências {ref=9.1}, {ref=9.2} e {ref=9.3}). Trata-se duma forma de mais facilmente referir um caso de uso e permite ao longo do projecto criar mecanismos de relacionamento mais simples entre os vários modelos que se vão construindo, nomeadamente quando se transformam os casos de uso em objectos.

Assim que o diagrama de casos de uso estiver desenhado, deve a equipa de projecto produzir uma descrição textual (ou gráfica) sobre cada um dos casos de uso, para servir como referência às fases seguintes do projecto. Há vários formatos possíveis para documentar a descrição dos casos de uso: texto informal, texto estruturado em passos numerados (com pré e pós-condições), pseudo-código ou diagramas de actividade [Schneider e Winters, 1998, pág. 37] [Booch et al., 1999, pág. 224].

Com base nos casos de uso do sistema, é possível seguir, no projecto, uma abordagem guiada ao risco. Para tal, deve construir-se uma lista com todos os casos de uso e atribuir-se a cada um deles um nível de importância, usando uma escala apropriada (por exemplo, com os seguintes 4 níveis: indispensável, importante, útil e supérfluo). Pode, assim, planear-se a implementação futura do sistema com base no nível de importância atribuído a cada caso de uso. Os casos de uso indispensáveis são implementados primeiramente, os importantes a seguir, os úteis depois e, se houver tempo, no fim, implementam-se os casos de uso supérfluos. Neste pressuposto, quando se ultrapassa o prazo estabelecido inicialmente para um projecto, é mesmo assim possível apresentar um sistema com as funcionalidades mais importantes a funcionar (e supostamente testadas). As funcionalidades não incluídas por não serem tão relevantes, podem ser introduzidas na versão seguinte do sistema, sem que daí resultem grandes inconvenientes para os utilizadores.

5.4 Diagramas de objectos

Assim que o comportamento exteriormente visível do sistema estiver definido, o analista deve identificar os objectos e as classes (e respectivas relações) que permitem descrever o sistema em desenvolvimento. Como ponto de partida para a análise orientada ao objecto, devem usar-se os diagramas de contexto e de casos de uso construídos anteriormente.

Como referido por Meyer [Meyer, 1988, pág. 52] e por Szyperski [Szyperski, 1998, pág. 8], uma década depois, ainda há muita gente que não distingue claramente entre os objectos e as classes. A confusão entre objectos e classes está profundamente relacionada com a natureza do software. Por exemplo, tanto a planta duma casa como a própria casa podem ser modeladas por objectos, mas também se pode entender a planta como a 'classe' da casa. Não há mal algum com esta visão enquanto os dois tipos de objectos estiverem separados. Os problemas surgem, no entanto, quando se misturam no mesmo modelo (ou diagrama) os objectos e as classes que dizem respeito aos sistemas a construir. Este é ainda um dos problemas mais comuns que se verificam, quando se usam conceitos orientados ao objecto [Sigfried, 1996, pág. xxiii].

Uma classe pode ser vista como um padrão que permite criar objectos para a aplicação em causa e, provavelmente, para outras aplicações futuras, enquanto que os objectos representam os elementos que constituem realmente a aplicação. Esta perspectiva permite concluir que é

preferível não incorporar no mesmo modelo as classes e os objectos, prática que é fortemente aconselhada no âmbito de MIDAS. Daí que, nesta subsecção, seja tratado o diagrama de objectos, deixando-se o diagrama de classes para a subsecção seguinte.

Para desenhar o diagrama de objectos, é necessário, primeiramente, identificar os objectos que compõem o sistema em desenvolvimento. Em MIDAS é usada uma estratégia própria para esta tarefa, baseada nos casos de uso. Fica, contudo, ao critério do analista escolher uma outra, caso a que aqui é sugerida não for do seu agrado.

5.4.1 Categorias de objectos

O espaço de análise pode dividir-se, como a fig. 5.4 ilustra, em três dimensões ortogonais: informação, comportamento e apresentação [Jacobson et al., 1992, pág. 131].

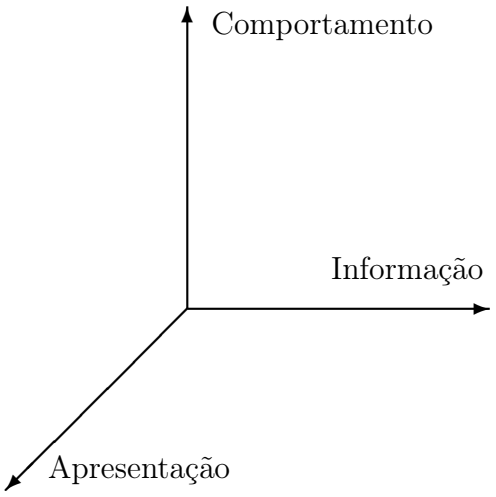


Figura 5.4: As 3 dimensões do espaço de análise.

A dimensão "informação" descreve a informação (dados) do sistema, o que permite especificar, implícita ou explicitamente, o estado interno do sistema. A dimensão "comportamento" especifica o comportamento que o sistema tem, i.e. indica quando e como o estado do sistema é alterado. A dimensão "apresentação" providencia os pormenores para apresentar o sistema ao exterior.

O modelo de análise constrói-se através da especificação de objectos neste espaço tri-dimensional. Uma hipótese, que não será adoptada neste trabalho, é usar objectos que expressam apenas uma única dimensão. É isso que, de alguma forma, se passa nos métodos estruturados, em que as funções são colocadas no eixo comportamento e os dados no eixo informação. Outra hipótese, que também não foi prosseguida neste trabalho, consiste na utilização de objectos, que podem ser colocados em qualquer posição do espaço (i.e. podem conter qualquer combinação das três dimensões). Esta perspectiva, adoptada na maioria dos métodos de análise orientada ao objecto, vê todos os objectos como iguais (todos eles têm atributos, operações e identidade própria), mas não permite classificar um objecto segundo a função (ou o papel) que desempenha no sistema.

Em MIDAS, adoptou-se a perspectiva que associa, a cada objecto encontrado na fase de análise, uma dada categoria, permitindo, desse modo, obter uma estrutura que pode, mais facilmente,

adaptar-se às mudanças [Jacobson et al., 1992, pág. 132]. Cada uma das categorias está fortemente relacionada com uma das dimensões, sem contudo implicar a inexistência de componentes nas outras 2 dimensões. Assim, os objectos podem classificar-se em três categorias principais:

- Objectos-interface.
- Objectos-entidade.
- Objectos-função.

Um *objecto-interface* modela comportamento e informação que dependem do interface do sistema, i.e. do diálogo (da comunicação) do sistema com os actores que com ele interagem. Tudo o que respeita a interface do sistema deve ser colocado em objectos-interface. O objecto-interface deve estar de tal forma encapsulado que, se houver alguma mudança no modo de comunicação, só o objecto-interface é modificado, ficando inalterados todos os restantes objectos. Os objectos-interface, por permitirem isolar as partes de interface das partes funcionais, tornam mais directa a reutilização destas (sob a forma de objectos-função).

Um *objecto-entidade* modela principalmente informação, cuja existência deve ser prolongada (não se incluem, portanto, dados com um carácter temporário). Para além dos atributos que caracterizam o objecto-entidade, todo o comportamento associado à manipulação dessa informação deve ser incluído nesse objecto-entidade. Como exemplo dum objecto-identidade considere-se uma conta bancária com os respectivos atributos e operações.

Um *objecto-função* modela comportamento que não pode ser associado, duma forma natural, a nenhum outro objecto, como, por exemplo, a funcionalidade que opera sobre vários objectos e que devolve o resultado a um objecto-interface. Considere-se um objecto que calcula o total dos saldos dum conjunto de contas bancárias. Esta funcionalidade poderia ser atribuída a um dos objectos-entidade (conta bancária) ou ao objecto-interface encarregado de apresentar o resultado final, mas não é, de facto, responsabilidade de qualquer um desses objectos. Daí que a solução mais apropriada, para exemplos similares a este, seja a introdução dum objecto-função responsável por disponibilizar essa funcionalidade.

Esta tipificação dos objectos deve ser considerada natural, pois um objecto unifica numa única entidade, ele próprio, os modelos (informação e comportamento) tradicionalmente divididos na abordagem estruturada. Neste sentido, é compreensível que, em algumas situações, um dado objecto tenha uma maior predominância sobre uma das dimensões expressas na fig. 5.4. Repare-se no recurso à expressão "*maior predominância*" que indica, por exemplo, que um objecto-entidade, além dos respectivos atributos, tem também associadas operações e que um objecto-função tem, além das operações, atributos, pois, afinal, ambos são objectos. Este foi o motivo principal que levou o autor a adoptar, para as categorias de objectos, nomes distintos daqueles usados para as dimensões, de forma a realçar que um dado objecto raramente está localizado num eixo.

A maioria das metodologias orientadas ao objecto não faz qualquer tipo de distinção relativamente à categoria a que um dado objecto pertence. É o que acontece, por exemplo, nas metodologias HOOD [Robinson, 1992], OMT [Rumbaugh et al., 1991] e Booch'91 [Booch, 1991]. Contudo, alguns autores fazem também uma classificação dos objectos, como aqui foi sugerido, apresentando-se, de seguida, algumas dessas propostas.

Sigfried sugere a utilização de apenas 2 categorias de objectos: objectos-informação e objectos-sistema [Sigfried, 1996, pág. 156]. Os primeiros são usados, principalmente, para modelar informação (correspondem exactamente aos objectos-entidade) e os segundos para suportar o

comportamento do sistema (correspondem a objectos-função ou a objectos-interface)[3].

A metodologia MOOSE distingue 2 categorias distintas de objectos, a que dá os nomes de *real world objects* e *policy objects* [Morris et al., 1996, pág. 110]. Os primeiros correspondem à conjugação de objectos-entidade com objectos-interface (i.e. modelam a informação de entidades reais e o interface com essas mesmas entidades), enquanto que os últimos são muito semelhantes aos objectos-função, uma vez que encapsulam comportamento relativo a vários *real world objects*. Na metodologia MOOSE, também se distinguem 3 categorias de objectos (*uncommitted objects*, *primitive objects*, *library objects*), mas essa classificação é feita segundo uma perspectiva complementar à anterior, baseada na complexidade dos objectos[4].

ROOM considera os objectos como passíveis de serem classificados em 2 categorias: *actors* e *data objects* [Selic et al., 1994, pág. 77]. Os primeiros correspondem a objectos activos, ao passo que os últimos correspondem à perspectiva mais tradicional dos objectos como instâncias de tipos abstractos de dados.

As categorias de objectos, adoptadas neste trabalho, tornam o modelo mais estável, na medida em que as mudanças a realizar, durante o desenvolvimento do sistema, são mais localizadas e afectam um número mais reduzido de objectos (de preferência apenas um). As alterações mais frequentes a que um sistema está sujeito são a sua apresentação e o seu comportamento, sendo a informação a componente mais estável dos sistemas (é este, afinal, o pressuposto fundamental em que assenta todo o paradigma dos objectos). Assim, mudanças à apresentação afectam apenas objectos-interface, enquanto que modificações ao comportamento podem perturbar qualquer tipo de objecto:

- Se a funcionalidade está associada a uma informação do sistema, então o objecto-entidade que a representa é afectado.
- Quando a funcionalidade a modificar está relacionada com a forma de apresentação, então o respectivo objecto-interface é alterado.
- Alterações a funcionalidades que envolvem vários objectos são, em princípio, locais a um objecto-função.

Uma vez mais, recorre-se ao mecanismo de estereótipos de UML para etiquetar os objectos segundo as 3 categorias consideradas, dado que esta categorização não foi considerada em UML. A fig. 5.5 mostra os estereótipos usados, em Midas, para as 3 categorias de objectos consideradas[5].

5.4.2 Construção do diagrama de objectos

O diagrama de objectos é construído a partir do diagrama de casos de uso, em que cada um destes é dividido em objectos das 3 categorias consideradas. Note-se que os casos de uso especificam a funcionalidade do sistema (i.e. dividem-no funcionalmente), ao passo que o diagrama de objectos tem associada uma estruturação que se pretende o mais robusta possível e que será

[3]Sigfried considera também objectos-interface mas como uma especialização dos objectos-sistema.

[4]Na realidade, são apresentadas 5 categorias de objectos, mas uma delas corresponde ao conceito de actor (entidade externa) e duas delas (*uncommitted objects* e *system objects*) são, na sua essência, a mesma categoria, daí só se terem considerado, na discussão, 3 categorias distintas.

[5]Os estereótipos ≪interface≫ e ≪control≫, definidos em UML e que se aplicam a classes e, consequentemente, a objectos, permitem tipicar um objecto como sendo essencialmente usado para efeitos de apresentação ou de comportamento, respectivamente. Portanto, correspondem aos objectos-interface e objectos-função definidos neste trabalho.

Categoria de objecto	Estereótipo
Entidade	≪entity≫ ≪data≫
Função	≪function≫ ≪control≫
Interface	≪interface≫

Figura 5.5: Estereótipos propostos para descrever as categorias de objectos.

a base para as fases de concepção e implementação. O diagrama de objectos representa uma arquitectura ideal para o sistema, já que, na sua construção, não foram considerados quaisquer factores relacionados com a plataforma de implementação (linguagem de programação, sistema operativo, processador, etc.).

A transformação ou, se se preferir, a transição do diagrama de casos de uso para o diagrama de objectos é um dos passos mais importantes e críticos da metodologia MIDAS, pois é nela que a arquitectura do sistema começa a ganhar forma. Trata-se duma tarefa que exige alguma criatividade por parte do projectista e para a qual não é possível fornecer regras precisas sobre como proceder. Esta transição consiste em distribuir o comportamento especificado pelos casos de uso, por objectos que serão os constituintes do diagrama de objectos. A metodologia REAL-TIME UML descreve um conjunto de 11 técnicas extremamente válidas para a identificação dos objectos que constituem um sistema [Douglass, 1998, pág. 94], mas nenhuma delas permite directamente transitar dos casos de uso para os objectos. Esta situação parece estar muito melhor resolvida em OOSE.

Embora não se imponha nenhuma estratégia em particular para a tarefa de identificação dos objectos, serão dadas, de seguida, algumas recomendações para o modo de proceder nesta tarefa. A primeira solução seria considerar cada caso de uso como um objecto, em consonância, aliás, com a seguinte afirmação:

> "*A use case may be viewed as an object*" [Jacobson et al., 1992, pág. 128].

Contudo, se assim se procedesse, não se facilitaria, como mais à frente se pretende mostrar, a localização das alterações que inevitavelmente se têm de realizar durante o ciclo de vida dum sistema (seja durante o desenvolvimento, seja depois aquando da utilização).

Uma outra alternativa, que igualmente não se subscreve neste trabalho, será usar objectos-entidade para armazenar apenas informação e colocar todo o comportamento dinâmico em objectos-função. Esta solução é de evitar, pois assim está a criar-se uma estrutura idêntica àquelas que resultam da aplicação de metodologias estruturadas, em que é evidente a divisão entre funções e dados e de que resultam os problemas indicados na secção 2.3.1. Assim, para precaver esses problemas, recomenda-se, entre outros, a associação de comportamento a objectos-entidade.

A solução preconizada pela metodologia MIDAS foi inspirada nas recomendações da metodologia OOSE, descritas por Jacobson *et al.*, a quem se atribui, justamente, a paternidade da ideia em utilizar, no âmbito das metodologias orientadas ao objecto, os casos de uso. A estratégia,

que constitui um dos contributos inovadores da metodologia, passa pela distribuição, a vários objectos, do comportamento especificado pelo diagrama de casos de uso, indicando explicitamente que parte do comportamento dum caso de uso corresponde a cada objecto introduzido no diagrama dos objectos. Note-se que é possível e até desejável que um mesmo objecto seja comum a diversos casos de uso.

Apesar de se permitir e até se incentivar alguma interacção no processo de passagem dos diagramas de casos de uso para os diagramas de objectos, sugere-se, por simplicidade de explicação, a seguinte ordem na identificação de objectos, segundo a respectiva categoria:

1. Objectos-interface.
2. Objectos-entidade.
3. Objectos-função.

Uma vez que os objectos são obtidos a partir casos de uso, as referências numéricas dos casos de uso devem ser transpostas para os objectos. A referência dum objecto constrói-se com a referência do caso de uso a partir do qual esse objecto foi criado, acrescentando um sufixo 'i', 'e' ou 'f', conforme se trate, respectivamente, dum objecto-interface, entidade ou função[6]. Desta forma, fica muito facilitada a observação das relações entre os casos de uso e os respectivos objectos, o que se revela importante para obter a continuidade dos modelos. Por outro lado, durante o projecto, verificou-se que as referências ajudavam não só nesse relacionamento entre modelos, como eram ainda muito úteis para designar os casos de uso e os objectos. Trata-se dum mecanismo que o autor aconselha vivamente a ser usado em projectos que envolvam um número considerável de elementos de modelação (sejam eles quais forem).

A ordem de identificação dos objectos pode ser aplicada a cada caso de uso sequencialmente ou a todos os casos de uso, em simultâneo. Na primeira estratégia, escolhe-se um caso de uso e identificam-se, por ordem, os respectivos objectos-interface, objectos-entidade e objectos-função. Quando estiverem identificados todos os objectos para um caso de uso, repete-se o processo de identificação para outro e assim sucessivamente, até todos os casos de uso estarem tratados. Alternativamente, pode escolher-se uma estratégia diferente, identificando, em primeiro lugar, os objectos-interface para todos os casos de uso, seguida da identificação de todos os objectos-entidade e concluída pela identificação de todos os objectos-função. Refira-se ainda que o facto de alguns objectos poderem ser comuns a vários casos de uso, vem apenas reforçar a necessidade dum processo não sequencial para a identificação dos objectos.

No âmbito do grupo de investigação do autor, foi igualmente concebida uma outra estratégia para transformar os casos de uso em objectos, tendo o autor participado na sua formalização (a ideia original é da autoria de Ricardo J. Machado). Essa estratégia encontra-se dividida em 4 passos [Fernandes et al., 1999]:

Passo 1: Transformar cada caso de uso em 3 objectos (1 objecto-função, 1 objecto-entidade e 1 objecto-interface). Cada objecto recebe a mesma referência que o caso de uso que lhe dá origem, acrescentando um sufixo que distingue a categoria a que pertence.

Passo 2: Com base nas descrições textuais, para cada caso de uso, decide-se quais as categorias que devem ser mantidas para representar computacionalmente o caso de uso. A decisão deve ser feita tomando em consideração todo o sistema (*abordagem holística*) e não cada caso de uso *per si*, como sucede em algumas abordagens mais reducionistas.

Passo 3: Os objectos "sobreviventes" devem ser agregados, sempre que houver sobreposição, numa representação unificada desses objectos.

[6]Por questões de compatibilidade com outras abordagens, permite-se também que se usem os sufixos 'd' (*data*) e 'c' (*control*), para objectos-entidade e função, respectivamente.

Passo 4: Os objectos e os agregados obtidos devem ser ligados para indicar as associações entre esses objectos.

5.4.3 Objectos-interface

Toda a funcionalidade dum caso de uso que estiver relacionada com o diálogo com um actor deve ser atribuída a um objecto-interface. O papel dum objecto-interface é transformar sinais provenientes do actor em eventos para o sistema e, simultaneamente, traduzir eventos do sistema em algo que seja apresentável (leia-se legível, perceptível, compreensível) para o actor. Geralmente, a identificação dos objectos-interface não é difícil, havendo, basicamente, três estratégias para o conseguir:

1. Procurar na descrição do problema referências explícitas a estes objectos.
2. Partindo de cada um dos actores, identificar os objectos-interface necessários para esse actor comunicar com o sistema.
3. Ler as descrições dos casos de uso e retirar a funcionalidade relativa a questões de interface.

A segunda alternativa é, porventura, a mais natural e simples, em entender do autor, uma vez que cada actor necessita, para interagir com o sistema, dum interface (a que podem corresponder um ou vários objectos-interface). Admitindo que a identificação de actores é considerada uma tarefa simples (secção 5.1), há, relativamente ao processo de identificação dos objectos-interface, uma forte probabilidade de sucesso, caso esta estratégia seja adoptada.

A terceira estratégia também é do agrado do autor, porque, percorrendo as descrições dos casos de uso, permite ao analista identificar, com relativa facilidade, os objectos-interface. A única ressalva reside no facto de, muitas vezes, não se escreverem as descrições dos casos de uso, o que inviabilizaria a utilização desta hipótese.

A primeira estratégia não é tão atractiva como as outras duas, porque obriga a reler a descrição do problema, que, como se sabe, é um documento como inúmeras ambiguidades, inconsistências e omissões.

Para identificar os actores, fez-se a divisão entre actores humanos e actores não humanos. De igual forma, é também possível separar os objectos-interface nessas duas categorias, uma vez que são esses objectos que comunicam directamente com os actores. Os objectos-interface que interagem com outros sistemas (não humanos) fazem-no normalmente através de sinais, segundo um dado protocolo de comunicação. Os objectos-interface que comunicam com utilizadores humanos requerem, habitualmente, interfaces gráficas.

5.4.4 Objectos-entidade

Tipicamente, um objecto-entidade modela informação que é necessária ao sistema durante um longo período de tempo, ou seja, essa informação deve ser guardada para além da existência do caso de uso. Neste sentido, informação criada e, posteriormente, destruída, durante a existência do caso de uso, não deve ser modelada como um objecto-entidade.

Um objecto-entidade corresponde normalmente a algum conceito real que existe fora do sistema e para o qual é necessário guardar alguma informação. Um exemplo é a modelação dum sistema de gestão de contas bancárias, em que há que guardar informação sobre os clientes, que são actores do sistema, mas que têm de ser representados internamente.

Para guardar a informação, os objectos-entidade usam atributos, a que deve associar-se um tipo de dados (primitivo ou composto), que define o domínio de valores admissíveis para esse atributo.

Alguns autores consideram que é relativamente simples encontrar os objectos-entidade dum determinado sistema, sendo porém mais complicado identificar as operações e os atributos que cada objecto-entidade deve conter [Jacobson et al., 1992, pág. 188]. Por outro lado, em algumas situações, não é fácil decidir se uma dada informação deve ser modelada como um novo objecto (do tipo entidade) ou como um atributo dum objecto-entidade já existente.

Além da informação, ao objecto-entidade são atribuídas as funcionalidades que, natural e intrinsecamente, se relacionam com essa informação (pelo menos, serviços para criar e destruir o objecto-entidade, e para ler e modificar os respectivos atributos).

5.4.5 Objectos-função

Finalmente, depois de identificados os objectos-interface e os objectos-entidade, restam os objectos-função. Note-se que alguns casos de uso podem dar apenas origem a objectos-interface e a objectos-entidade, não sendo portanto necessário qualquer objecto-função. Embora esta seja uma situação muito frequente, para a maioria das situações, é importante, para outros casos de uso, encontrar os objectos-função que podem contemplar funcionalidades ainda não atribuídas nas outras duas categorias de objectos. Essa funcionalidade tem de ser colocada num objecto-função quando não puder ser associada, duma forma natural, a nenhum outro objecto das outras duas categorias.

A única recomendação a dar, para encontrar objectos-função, consiste em detectar quais os casos de uso que ainda não estão completamente incluídos em objectos-interface ou em objectos-entidade.

É essencial atender ao facto que o propósito com que se modela um dado sistema pode dar origem a objectos de categorias distintas. O mesmo componente pode ser modelado por um objecto-entidade ou por um objecto-função. Por exemplo, considere-se um sensor, que serve para medir uma dada grandeza física. Se o propósito de modelação é um sistema de inventário de componentes, então o sensor dá origem, no modelo, a um objecto-entidade, onde são guardados, entre outros, a marca, o preço, o intervalo de valores considerado e a precisão. Se o sistema a construir necessita dum sensor como uma das suas partes constituintes, para obter o valor da grandeza associada, então ter-se-á, no modelo, um objecto-função.

Tipicamente, um objecto-entidade tem uma estrutura mais simples que um objecto-função. Por exemplo, um objecto-entidade pode conter primordialmente atributos e operações simples para os manipular, enquanto que um objecto-função contém internamente outros sub-objectos. Além disso, um objecto-função é usualmente activo, interage com os objectos-interface e usa os objectos-entidade para ler e guardar informação.

Outra forma de distinguir entre objectos-entidade e objectos-função consiste em tentar verificar quantas instâncias da mesma classe existem no modelo. Se forem muitas, então trata-se, com grande probabilidade, de objectos-entidade (exemplo: contas bancárias ou peças em inventário). Em contraponto, quando só for necessário uma única instância duma classe, é provável que se esteja perante um objecto-função. Contudo, neste caso, pode também tratar-se dum objecto-entidade em que é necessário apenas uma instância (exemplo: configuração do sistema).

5.4.6 Composição de objectos

Quando o número de objectos a colocar no diagrama ultrapassa um dado valor, o que é vulgar em sistemas complexos, a legibilidade do diagrama começa a degradar-se significativamente. Recorde-se a famosa regra "7 ± 2" que indica que a mente humana não consegue, em simultâneo, captar adequadamente mais do que 7 conceitos relacionados (com uma margem, para cima ou para baixo, de 2) [Miller, 1957]. Atente-se no emprego da expressão "*em simultâneo*". Esta inaptidão, para manipular várias coisas simultaneamente, consiste numa limitação intrínseca à maioria dos seres humanos que não pode ser eliminada através de aprendizagem. Para colmatar esta falha, pode e deve procurar-se algum mecanismo de modelação que permita formar novos objectos, a partir dum conjunto de objectos de mais baixo nível.

Nesta linha de raciocínio, são necessários mecanismos que facilitem a criação de objectos compostos (criando relações entre objecto/sub-objectos, agregado/partes ou sistema/subsistemas), para permitir especificações do mais abstracto para o mais concreto. Em muitas situações, é conveniente tratar um dado conjunto de objectos como uma única unidade conceptual que faz parte do sistema em desenvolvimento. Em sistemas orientados ao objecto, a solução passa pela introdução dum objecto agregado ou composto que constitui uma abstracção, que esconde, a um dado nível, os seus sub-objectos e as respectivas relações.

Visto do exterior, um objecto composto é igual a outro qualquer objecto. Consequentemente, pode ser incorporado como parte da estrutura de um outro objecto composto. Este processo de incorporação de objectos pode ser aplicado a qualquer nível de profundidade, o que permite construir sistemas com estruturas hierárquicas complexas. Esta possibilidade permite que um dado sistema seja incorporado num outro sistema de maior âmbito (abordagem ascendente), ou que um sistema seja construído sem conhecer a sua estrutura interna (abordagem descendente).

A introdução deste tipo de objectos, que, além das características habituais dum objecto (nome, atributos e operações), inclui também uma estrutura que define os objectos que o compõem e as respectivas interligações, obriga a repensar a forma como especificar as classes de modo a considerar esta questão. A solução adoptada em MIDAS passa por considerar a estrutura de um objecto, como mais um atributo que lhe está associado. Alternativamente, poderia adoptar-se a solução de Sigfried que propõe que uma classe tenha além do nome, atributos e operações, uma quarta secção que permite especificar a estrutura interna dos objectos.

A composição também é referida como agregação não partilhável (*non-overlapping aggregation*). O uso de agregação em que há partilha de partes por vários agregados deve evitar-se porque não facilita o relacionamento entre a realidade e o modelo construído [Sigfried, 1996, pág. 91]. Assim, para colmatar este problema, deve tentar dividir-se os objectos que pertencem a vários compostos segundo vistas complementares. O seguinte exemplo pretende demonstrar esta ideia. A fig. 5.6(a) apresenta uma série de objectos todos eles representando divisões duma casa, estando agregados segundo o aquecimento instalado e as canalizações de água disponíveis. Como o quarto de banho e a cozinha necessitam de água e aquecimento, essas divisões têm que ser partilhadas. Uma solução para esta partilha passa por modelar o quarto de banho segundo duas perspectivas distintas (água e aquecimento), dando origem a dois objectos distintos. Procedendo de igual modo relativamente à cozinha, obtêm-se os agregados mostrados na fig. 5.6(b).

Em vez de um objecto representar vários aspectos, têm-se vários objectos representando cada um deles uma vista distinta do sistema, evitando assim a necessidade de partilhar objectos entre agregados. O uso da composição tem como vantagem adicional permitir a representação

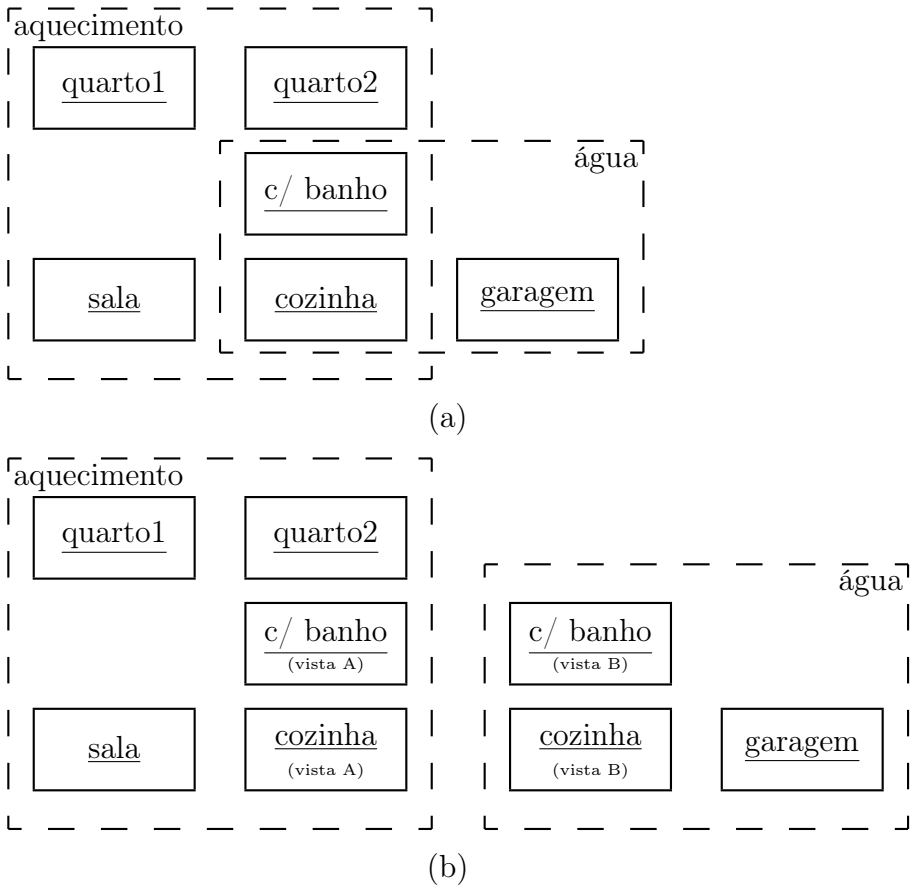


Figura 5.6: (a) Agregação, em que há partilha de componentes; (b) Agregação, em que não há partilha de componentes, mas em que estes são representados por vários objectos, focando cada um deles uma dada vista.

da relação entre composto e componentes de modo intuitivo (por inclusão gráfica), o que reduz a quantidade de informação a captar para perceber o modelo.

5.4.7 Objectos repetidos

Por vezes, é necessário especificar múltiplos objectos da mesma classe, como por exemplo, quando se tem de construir uma estrutura repetitiva (tipo rede neural artificial ou *array* de processadores). Por este motivo, a notação UML foi estendida para suportar este mecanismo de especificação, considerado útil em MIDAS. Deste modo, com um único símbolo, designado por *objecto repetido*[7], representa-se uma série de objectos organizados duma forma estruturalmente regular. Trata-se, portanto, dum mecanismo de abstracção, em que o objecto é replicado várias vezes.

Assim, sugere-se, para representar um objecto repetido, a notação mostrada na fig. 5.7(a), em que o número de objectos, neste caso 4, é indicado no canto superior direito. Esse valor representa o número de instâncias nesse contexto. Na fig. 5.7(b), para cada um dos 2 aparelhos, existem 4 luzes e 4 botões.

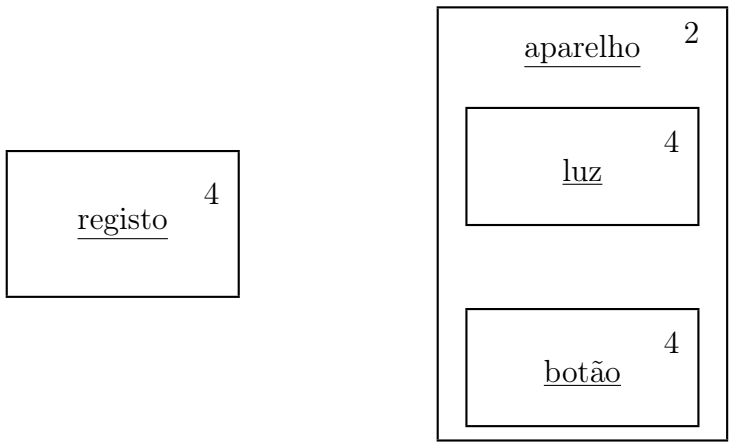


Figura 5.7: Notação UML para representar objectos repetidos.

A replicação dum objecto implica também a replicação das suas entradas, das suas saídas e dos seus objectos componentes. Assim, são necessárias formas para expressar as ligações que se fazem entre os objectos que o objecto repetido pressupõe e os objectos que com eles comunicam.

Em relação aos sub-objectos assume-se que cada objecto (dum objecto repetido) é composto por todos os sub-objectos indicados no objecto repetido. Quanto às ligações, as combinações consideradas restringem-se às seguintes hipóteses:

- Ligação entre um objecto repetido e um objecto.
- Ligação, um a um, entre dois objectos repetidos.
- Ligação, todos com todos, entre dois objectos repetidos.

O exemplo da fig. 5.8 mostra essas 3 hipótese, em que se consideram 2 objectos repetidos (objA e objB), cada um dos quais representando 3 objectos, e um objecto simples (objC). Usaram-se

[7] Quando for necessário distinguir entre um objecto repetido e um não repetido, usar-se-á o adjectivo "simples" para qualificar este último.

unicamente interacções a ligar os objectos, mas genericamente poderia usar-se qualquer das interligações inter-objectos apresentadas na secção 5.2.

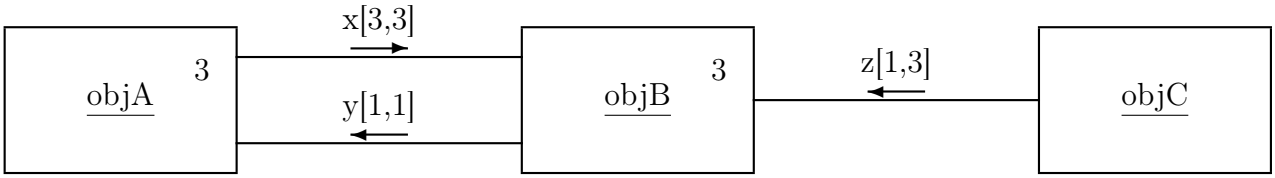


Figura 5.8: Ligações envolvendo objectos repetidos.

Pressupondo que cada objecto repetido dá origem a 3 objectos com os índices escritos em forma de *array*, a fig. 5.8 pode ser refeita de forma a obter a fig. 5.9 em que se especificam individualmente os objectos e as ligações.

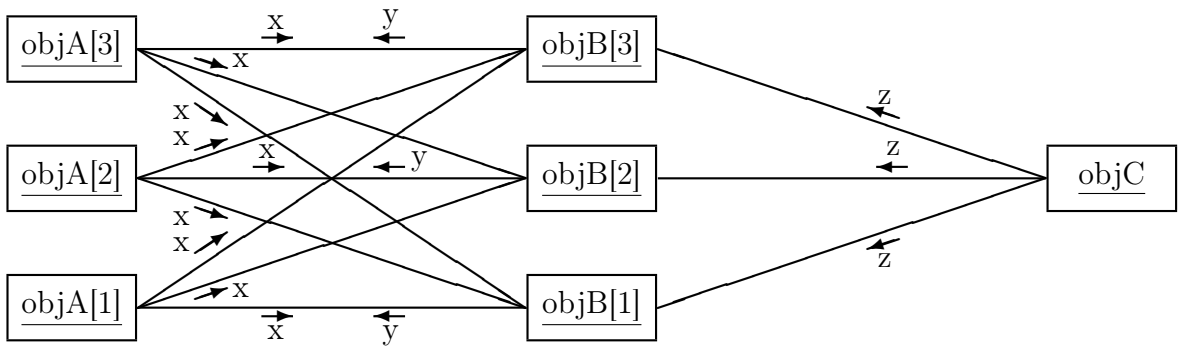


Figura 5.9: Ligações pormenorizadas entre os objectos da figura anterior.

Qualquer dos 3 objectos objA pode enviar uma mensagem x a qualquer dos 3 objectos objB, pelo que essa ligação entre objectos repetidos equivale a 9 ligações entre os objectos que representam (ligação todos com todos). Cada um dos objectos ObjB pode enviar uma mensagem y ao respectivo objecto ObjA: o objecto objB[1] envia ao objecto objA[1], o objecto objB[2] envia ao objecto objA[2] e o objecto objB[3] envia ao objecto objA[3].

Ligações menos regulares que estas (por exemplo, a ligação do objecto objC apenas ao objecto objB[3]) terão de ser explicitamente referidas no diagrama, através, por exemplo, duma nota de texto.

5.4.8 Criação e eliminação dinâmicas de objectos

Um mecanismo muito comum, em modelos de análise para sistemas software, consiste na possibilidade em criar e eliminar dinamicamente objectos. Daí que, em software, a implementação deste mecanismo não apresenta qualquer tipo de dificuldade. A implementação de objectos criados dinamicamente, embora seja possível ser feita em hardware, devido aos últimos avanços da lógica reconfigurável [Sklyarov et al., 1998], apresenta ainda inúmeras questões em aberto, por se tratar duma tecnologia pouco amadurecida. Assim, embora não seja de descartar esta possibilidade, o seu estudo não foi considerado prioritário no actual contexto de MIDAS.

5.4.9 Introdução do tempo

Em relação às questões temporais, completamente essenciais em sistemas embebidos de tempo-real, existem dois tipos de informação que um sistema pode necessitar. Por um lado, podem ser precisos sinais periódicos para cumprir as suas tarefas, como, por exemplo, para registar periodicamente uma dada informação em historial (vulgo *log*). Por outro, pode ser necessária a informação do tempo absoluto para determinar quando devem executar-se algumas das funcionalidades do sistema, por exemplo, para determinar quando se deve proceder à rega dum campo agrícola (ligar às 19h20m e desligar às 20h35m).

Existem duas formas alternativas de introduzir o tempo como parte do sistema em desenvolvimento [Schneider e Winters, 1998, pág. 18]. Uma das formas consiste em considerar um actor tempo, responsável por iniciar os casos de uso que carecem de informação temporal. Dado que os actores são entidades localizadas fora da periferia do sistema em causa, o projectista, caso adopte esta solução, deixa de poder definir e controlar a forma como o tempo é tratado. Alternativamente, pode considerar-se que o tempo está incluído no interior do sistema, a que corresponde um objecto interno ao sistema que faculta a informação de cariz temporal.

Em MIDAS, a solução para abarcar estas questões de carácter temporal, passa então por esta segunda alternativa, através da introdução dum objecto-função, que disponibiliza a informação temporal a todos os outros objectos que dela carecem. Para mais facilmente identificar este objecto, propõe-se a aposição do estereótipo ≪timer≫ que especializa a categoria do objecto (≪timer≫ como uma sub-categoria de ≪function≫). Os requisitos desse objecto podem ser identificados, depois de determinar os objectos que dele dependem e o tipo de informação de que necessitam (resolução do tempo, tempo relativo ou absoluto, informação periódica ou esporádica, etc.).

5.5 Diagramas de classes

A maioria das metodologias orientadas ao objecto propõe a construção dum diagrama de classes e só, em alguns casos, sugere que posteriormente seja construído o diagrama de objectos. Em MIDAS sugere-se exactamente que essa ordem seja trocada. É entender do autor que, para o desenvolvimento dum sistema embebido, um bom diagrama de objectos é mais importante que um bom diagrama de classes, uma vez que são os objectos que, ao fim e ao cabo, constituem o sistema. Por este motivo, a estratégia trilhada, na fase de análise, é iniciada com a identificação dos objectos seguida da sua classificação, ou seja, da escolha das classes a que esses objectos pertencem. Não se está, de maneira alguma, a advogar que o diagrama de classes deva ser superficialmente trabalhado ou até negligenciado (a situação desejável é obviamente ter bons diagramas de objectos e de classes), mas antes que a ênfase deverá ser, primeira e predominantemente, colocada no diagrama de objectos. A seguinte afirmação mostra que esta postura tem a sua justificação e que a habitual ênfase colocada na construção da estrutura de classes pode ser relaxada:

> "*(...) developers concentrate too much on the class structure and too little on the object structure.*" [Sigfried, 1996, pág. 146].

É possível que esta visão, que coloca as classes num papel aparentemente secundário, possa ser classificada, por alguns especialistas no paradigma dos objectos, como baseada em objectos

(e não orientada ao objecto). No entanto, a abordagem que define primeiro os objectos e só depois as classes é, de alguma maneira, semelhante ao processo *bottom-up*, definido em OMT, para organizar as classes numa estrutura hierárquica [Rumbaugh et al., 1991, pág. 163].

Além disso, e não questionando as indiscutíveis importância e utilidade que as classes têm no processo de desenvolvimento, muitas metodologias, que se auto-intitulam orientadas ao objecto, começam a relegar a herança para um plano de menor importância, como é possível concluir-se da seguinte afirmação de Budgen:

> "*While there are those who consider the concept of inheritance as central to the object-oriented paradigm, many current 'object oriented' design practices are only concerned with developing a design model by considering the first three properties [abstraction, encapsulation, modularity] of the 'object model', and with constructing a module hierarchy that is based largely on the uses relationship.*" [Budgen, 1994, pág. 274].

Assim, parece perfeitamente válido que só depois de identificados os objectos que compõem o sistema, se proceda à sua classificação, i.e. se determine a classe a que pertence cada um desses objectos. É durante esta tarefa que a estrutura das classes é construída, alterada ou, idealmente, apenas usada e onde a reutilização tem a sua expressão mais vincada, segundo três vertentes. Primeiramente, caso exista mais do que um objecto da mesma classe, a definição desses objectos é feita num único local. Em segundo lugar, caso se descubram classes com características comuns, podem criar-se relações hierárquicas entre essas classes. Finalmente, ao identificar a classe dum objecto, pode reconhecer-se que ela já existe numa biblioteca, o que permite a sua (re)utilização imediata.

O diagrama de classes é um referencial para uma categoria de aplicações que a partir dele se podem construir, ou seja, o modelo de classes é uma generalização de alto nível dos sistemas [Lyons, 1998]. Dito por outras palavras, quando se define a forma como as classes se relacionam entre si, está a indicar-se quais os possíveis sistemas (ou, segundo uma perspectiva diferente, todos os estados ou configurações dum dado sistema) que podem ser construídos a partir dessas classes. Neste sentido, pode afirmar-se que o diagrama de classes funciona como uma "camisa de forças" para o diagrama de objectos.

Daí que é comum, em muitas metodologias, não se proceder à construção do diagrama de objectos, uma vez que este resulta automaticamente do diagrama de classes. Nas situações em que a construção do diagrama de objectos é feita, é necessário garantir que as relações expressas, no diagrama de classes, entre duas classes, se verificam igualmente entre instâncias dessas classes. Por este motivo, nas metodologias que preconizam a existência de diagramas de classes e objectos, é habitual aqueles serem construídos em primeiro lugar.

Além disso, há uma tarefa adicional em que se tem de assegurar a coerência entre a informação que consta nos diagramas de objectos e classes, o que pode ser interpretado como um sintoma de que há informação que está a ser replicada desnecessariamente [Douglass, 1998, pág. 130]. O facto de existir um estereótipo ≪singleton≫ em UML, que indica que uma dada classe terá uma e uma só instância é revelador desta visão que perspectiva o diagrama de classes como um padrão para os sistemas, numa dada categoria. Este estereótipo indica claramente que o diagrama de objectos a construir, para ser coerente com o diagrama de classes, tem de satisfazer essa restrição de só ter uma instância dessa classe ≪singleton≫.

Este tipo de abordagem parece ser muito popular para desenvolver, por exemplo, sistemas de informação em que os objectos que compõem o sistema são criados e destruídos durante a vida

útil deste. Por exemplo, num sistema de gestão de contas bancárias, é comum cada conta ter associado, pelo menos, um titular (este facto é indicado no diagrama de classes associando a classe conta à classe titular). Assim, sempre que se cria um objecto conta é necessário ligá-lo a, pelo menos, um objecto titular.

Na opinião do autor, esta abordagem não apresenta grandes benefícios, no desenvolvimento dum sistema embebido, pois habitualmente os objectos que o compõem não são criados e destruídos. Um sistema embebido é, regra geral, composto por um conjunto pré-determinado de objectos, cujas interligações não podem, à partida, ser definidas. Daí que não seja importante indicar que, por exemplo, os objectos da classe controlador estão ligados a objectos da classe sensor. Se numas aplicações, essa informação até possa ser pertinente, noutras não o será certamente. A pouca relevância dum diagrama de classes torna-se mais evidente quando se pretende construir um único sistema (para uma aplicação específica), o que é comum para muitos dos sistemas embebidos a desenvolver.

Nesta linha de raciocínio, a metodologia MIDAS vê o diagrama de classes apenas como um repositório de especificações pré-definidas de objectos ("um armazém de matéria prima"), que podem aproveitar-se para qualquer aplicação que se pretenda desenvolver. Nesse sentido, a estrutura de classes pode ser composta por várias árvores[8] não ligadas entre si, não se impondo portanto uma única estrutura para incluir todas as classes relevantes, como sucede, por exemplo, em SMALLTALK.

Em MIDAS, abraçou-se a perspectiva que preconiza, salvo raras excepções, as subclasses como semanticamente relacionadas com as suas superclasses. Esta perspectiva não inviabiliza a reutilização das implementações, que ocorre como uma consequência natural das relações semânticas entre classes. Apesar do mecanismo de herança não impor nenhuma perspectiva, adoptou-se a prática de ver as superclasses como mais abstractas que as subclasses, o que permite que a hierarquia de classes seja igualmente uma hierarquia de abstracções e que o mecanismo de herança seja um esquema de classificação, no verdadeiro sentido da palavra.

5.5.1 Composição e Associação

Habitualmente, os diagramas de classe apresentam uma série de relações entre classes. UML disponibiliza as seguintes relações: associação, agregação, composição, generalização e refinamento (fig. 3.11). Segundo o autor, trata-se dum exagero de relações, sendo até possível apenas utilizar a relação de herança, sem contudo diminuir a extrema utilidade do diagrama de classes. O uso das outras relações, nomeadamente a associação e a agregação, tem fortes raízes na área dos sistemas de informação, em que o diagrama de classes constitui, como se viu atrás, uma espécie de padrão para a forma como são estruturadas as aplicações obtidas a partir desse diagrama.

Considere-se a relação de composição entre classes e conclua-se se há realmente necessidade de incluir ou não esse tipo de relação no diagrama de classes. Considere-se, como exemplo, que se deseja construir um computador, que corresponde à aplicação a desenvolver. Para tal, usam-se vários componentes ou partes (por exemplo chassis, memórias, processador, disco, monitor e teclado), que são obtidos por instanciação de classes, sendo o computador construído pela composição desses componentes. Quando o computador está totalmente construído, é possível então vê-lo como um objecto, por si só, e que por tal deve ter a respectiva classe.

[8] Árvores e não grafos, uma vez que se parte do princípio que não se recorre ao mecanismo de herança múltipla, pelos motivos indicados mais à frente.

Este exemplo mostra que há uma noção temporal associada à construção do computador. Primeiramente, classes de baixo nível (como memória) devem estar disponíveis. Depois, a atenção é focada na estrutura de objectos, onde se constrói o sistema a partir de instâncias dessas classes. No fim, tenta novamente observar-se se, na estrutura de classes, faz sentido acrescentar a classe do objecto recém criado por composição. Se a conclusão for positiva, a estrutura de classes é actualizada com a inclusão de uma nova classe.

Nesta linha de raciocínio, incluir relações de agregação ou composição na estrutura de classes é inadequado (para não dizer errado), uma vez que essa informação é melhor modelada na estrutura de objectos. Em Midas, incluir na estrutura de classes, além de associações de hierarquia, relações de agregação ou composição é desaconselhado, por considerar-se que é conceptualmente confuso. A seguinte citação corrobora esta visão:

> "*(...) the class structure should be kept as clean from associations other than the is_a association as possible.*" [Sigfried, 1996, pág. 107].

Quanto à associação entre classes trata-se duma relação com fortes raízes na modelação de sistemas de informação (exemplo: homem "éCasadoCom" mulher), em que se vê o diagrama de classes como um referencial para as aplicações (diagramas de objectos). Em sistemas embebidos, e mesmo em sistemas software, este tipo de informação pode perfeitamente ser indicada apenas no diagrama de objectos, onde se explicita quais os objectos que realmente estão associados (exemplo: Bill "éCasadoCom" Hillary e Jorge "éCasadoCom" Maria José).

5.5.2 Classes concretas e abstractas

Como construir então uma estrutura de classes ligadas unicamente através de relações hierárquicas? Para dar resposta a esta questão recorrer-se-á ao *princípio de substituição de Liskov* (LSP) que afirma que uma subclasse deve responder a todas as mensagens que as suas superclasses também respondem, ou seja, se a classe B herda de A, então uma instância de B pode sempre ser usada em todos os contextos em que instâncias de A são utilizadas. Daqui resulta a impossibilidade de uma subclasse eliminar propriedades das respectivas superclasses, podendo contudo acrescentar ou especializá-las. Portanto, o LSP está intrinsecamente ligado ao uso de herança estrita. Deste modo, a hierarquia de classes contém as classes mais abstractas no topo e as mais especializadas em baixo. Este facto resulta da forma como a hierarquia é construída e não de nenhuma característica especial do mecanismo de herança, uma vez que é possível imaginar outras estruturações, mas nenhuma parece ser tão utilizável quanto a referida.

Uma *classe concreta* é formada com o propósito de criar directamente instâncias suas. Por contraponto, uma *classe abstracta* é uma classe que serve apenas para ser herdada por outras classes e que portanto não tem instâncias criadas a partir dela. Um classe abstracta está parcialmente definida, pois algumas das suas operações não estão implementadas, i.e. não existem os métodos que as implementam, sendo apenas indicada a interface das operações (o nome, os parâmetros e o tipo do resultado). É apenas nas subclasses concretas que essas operações são implementadas, uma vez que as respectivas instâncias necessitam obrigatoriamente de conhecer os métodos. Contudo, a maioria dos mecanismos de herança não restringe a possibilidade de se criarem instâncias a partir duma classe abstracta (ou, visto sob outra perspectiva, de se obterem subclasses a partir de classes concretas). Esta facilidade deve, no entanto, ser evitada uma vez que, desta forma, as subclasses ficam dependentes das particularidades da classe concreta. Se for necessário alterar as características desta, esse efeito repercute-se imediatamente

nas subclasses, o que nem sempre é desejável.

O seguinte exemplo pretende ilustrar esta problemática, com recurso às classes Fila e Pilha [Selic et al., 1994, pág. 259–61]. Considere-se que, inicialmente, existia a classe concreta Fila, que disponibiliza os métodos põeTopo e removeTopo. Posteriormente, pretendeu definir-se a classe Pilha como uma subclasse de Fila, que herda os métodos põeTopo (para *push*) e removeTopo (para *pop*) e acrescenta um novo método topo que devolve o elemento do topo da pilha sem o remover (fig. 5.10). Assuma-se que depois se verificou a necessidade de incluir novos métodos (põeFundo e removeFundo) na classe Fila. Como estes métodos são propagados automaticamente para as subclasses, passam a estar disponíveis em Pilha, violando-se nesta classe o princípio LIFO (o último a entrar tem que ser o primeiro a sair).

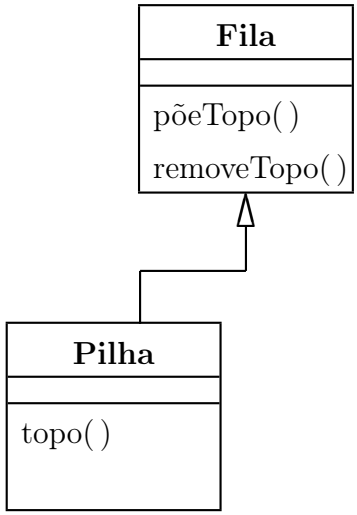


Figura 5.10: As classes Fila e Pilha.

Esta interdependência entre as duas classes pode ser evitada, se for criada uma classe abstracta comum, como a fig. 5.11 ilustra. Deste modo, se houver uma propriedade comum às duas classes que tenha de ser adicionada, a classe abstracta é o local onde a nova propriedade é indicada. Caso a propriedade seja apenas relativa a uma das classes concretas, a adição daquela far-se-á apenas na respectiva classe, deixando a outra inalterável.

Assim, sugere-se, no âmbito da metodologia MIDAS, que apenas as classes concretas possam ser folhas da estrutura hierárquica que o mecanismo de herança determina e que todas as superclasses sejam classes abstractas. Deste modo, garante-se que eventuais modificações (por exemplo, acrescentar mais um atributo) possam, caso se pretenda, apenas afectar uma classe. Nos diagramas de classes UML, a distinção entre classes abstractas e concretas, faz-se por aposição das restrições {abstract} e {instantiable}, respectivamente.

5.5.3 Herança estrita

Conceptualmente, a utilização de herança estrita deve prevalecer em relação à herança não estrita. Dado que, nesta última, uma subclasse pode modificar ou eliminar características da sua superclasse, deixa de se poder ver uma instância da subclasse como sendo também

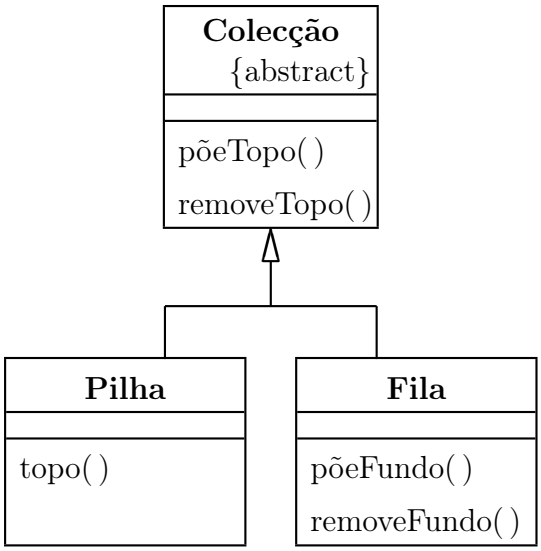


Figura 5.11: Reestruturação das classes Fila e Pilha, através da introdução duma classe abstracta.

instância da superclasse. Em Midas, a modificação ou eliminação de características herdadas não é aconselhada, embora para certas situações possa ser uma solução prática a considerar.

Pode dar-se a situação de existir uma classe que corresponda "quase" na totalidade àquilo que se necessita, mas que para ser reaproveitada necessita de uma reorganização profunda da hierarquia de classes. Neste caso, é preferível herdar-se dessa classe, acrescentando as propriedades em falta e modificando e eliminando outras herdadas. Deve, contudo, evitar-se a utilização de *herança por acidente*, apenas com o único objectivo de reutilizar (reaproveitar) código (atributos ou operações), esquecendo a relação hierárquica, ao nível conceptual, entre as classes. Um exemplo absurdo, mas que pretende ser ilustrativo, é a classe Gato herdar da classe Mesa (a relação inversa também seria absurda), pelo simples facto de os gatos e as mesas terem 4 pernas.

Repare-se que o princípio de substituição de Liskov só é realmente relevante quando há classes concretas que herdam de outras classes concretas, uma vez que as instâncias das subclasses mais especializadas podem ser vistas, nalgum contexto, como instâncias da classe mais genérica. Uma vez que, como indicado na subsecção 5.5.2, apenas as folhas da hierarquia de classes devem ser classes concretas, parece poder concluir-se que, em Midas, se pode usar herança não estrita entre todas as classes. Apesar dessa conclusão ser válida, não se preconiza essa possibilidade, ao nível da especificação, embora se admita a sua utilização em circunstâncias especiais como atrás se alvitrou.

5.5.4 Herança múltipla

A herança múltipla, usualmente, só faz sentido se os conjuntos herdados forem especializados, segundo características ortogonais (sem nada em comum). A possibilidade de utilizar relações de generalização do tipo *and* (que se designam *and-generalization* em UML), em que a especialização se faz simultaneamente ao longo de várias dimensões ortogonais, é de pouca aplicabilidade para sistemas de tempo-real (embebidos), uma vez que introduz demasiada complexidade [Douglass, 1998, pág. 143]. Daí não se considerar essa possibilidade neste trabalho, uma vez que, mesmo que essa condição seja satisfeita, o uso de herança múltipla não é acon-

selhado, embora seja possível dado que a linguagem OBLOG suporta esse mecanismo. A razão para este conselho reside numa ponderação entre os custos e as vantagens que resultam da inclusão do mecanismo de herança múltipla no meta-modelo subjacente às especificações do sistema. É entender do autor que os primeiros suplantam, duma forma evidente, as segundas, o que está de acordo com a seguinte afirmação de Rumbaugh *et al.*:

> "*Multiple inheritance may be used to increase sharing, but only if necessary, because it increases both conceptual and implementation complexity.*" [Rumbaugh et al., 1991, pág. 164].

A utilização de herança repetida é, nesta linha de raciocínio, completamente desaconselhada, por evidenciar ainda mais os custos. A *herança repetida* é um caso especial da herança múltipla e significa, na prática, que uma mesma subclasse herda da mesma classe várias vezes. A fig. 5.12 apresenta um exemplo de herança repetida, em que a classe clsD herda as propriedades de clsA por duas vias (por clsB e por clsC). Os problemas com este mecanismo de herança surgem, quando, por exemplo, o método mét1 definido em clsA é alterado (reescrito) em clsB. A seguinte pergunta é, nessa situação, mais do que óbvia: "*qual o método que é usado em clsD?*". Repare-se que há duas respostas para esta questão (o método definido em clsB, se a herança for feita via clsB, ou o método definido em clsA, se a herança for feita via clsC), pelo que algures no processo de desenvolvimento vai ser necessário resolver este conflito.

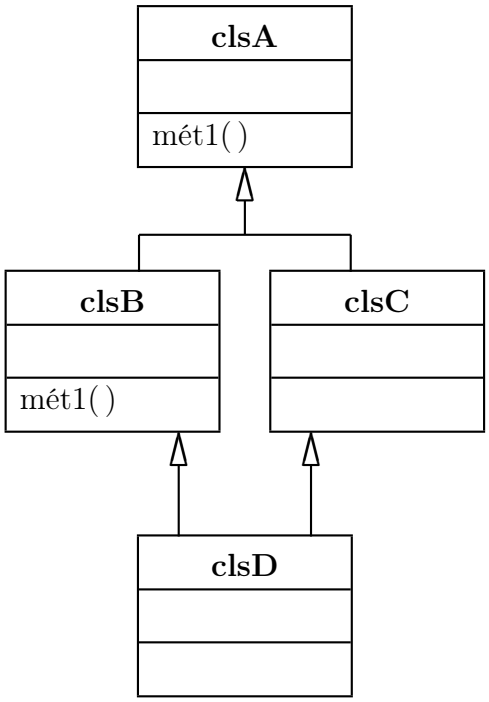


Figura 5.12: Exemplo do mecanismo de herança múltipla.

A necessidade do mecanismo de herança múltipla pode ser apartada ou mesmo eliminada, se existir a possibilidade de combinar vários componentes de forma a criar um novo objecto [Selic et al., 1994, pág. 165]. Como, em MIDAS, essa possibilidade está disponível, mais evidente se tornam as vantagens marginais que poderiam advir, caso o mecanismo de herança múltipla fosse considerado.

Note-se que uma subclasse não herda somente os atributos e as operações da superclasse. Tanto a estrutura (i.e. os componentes e respectivas interligações), como o diagrama de estados (em UML, usam-se diagrama de state-charts como modelos baseados em estados) são herdados, o que levanta mais algumas questões, que se abordam na secção seguinte.

5.6 Diagramas de state-charts

Até ao momento, definiu-se a forma como os sistemas devem ser decompostos nos seus objectos constituintes e o modo como estes se inter-relacionam, mas a análise orientada ao objecto inclui também a especificação do comportamento dinâmico dos sistemas, através da utilização de diagramas de state-charts. O *comportamento* dum sistema refere-se à operação interna deste ao longo do tempo [Selic et al., 1994, pág. 71]. Na metodologia Midas, de acordo com o tipo de comportamento que os objectos exibem, estes podem ser classificados segundo as duas seguintes categorias, todas eles consideradas relevantes para a área dos sistemas embebidos:

- simples/reactivo.
- passivo/activo.

Nesta secção, em relação aos state-charts, apresentam-se ainda algumas regras que devem ser respeitadas na sua construção, o respectivo modelo temporal e a forma como a herança deve ser reinterpretada devido à sua utilização.

5.6.1 Objectos simples e reactivos

Qualquer objecto cujo comportamento não dependa da história associada, diz-se que tem um *comportamento simples* e não tem, portanto, associada qualquer noção de estado. Alguns objectos-informação tem o seu comportamento enquadrado neste tipo. Por exemplo, um objecto que disponibiliza aos seus clientes o cálculo da raiz quadrada dum número, devolve sempre o mesmo resultado, independentemente da história (i.e. das raízes quadradas anteriormente calculadas).

Se um objecto tem um *comportamento reactivo*[9], então este pode ser descrito por um modelo à luz dum meta-modelo baseado em estados, nomeadamente por um state-chart que é o formalismo adoptado em UML. Desta forma, em qualquer instante, o objecto está num dado estado que determina a forma como aquele reage às mensagens a que pode estar sujeito. Note-se que o state-chart não especifica a funcionalidade duma operação, mas antes de todo o comportamento do objecto (o que potencialmente envolve muitas operações). Associadas às transições e aos estados dum state-chart podem estar chamadas a métodos definidos no espaço do respectivo objecto (mais rigorosamente, classe), pelo que o state-chart indica quando e como os métodos do objecto são invocados[10].

[9]Segundo a terminologia da área dos sistemas digitais, um sistema reactivo é classificado como sequencial, enquanto que um sistema simples é classificado como combinatório.

[10]O meta-modelo RdP-shobi tem uma semântica idêntica a esta, em que a Rede de Petri descreve o fluxo de controlo do sistema e os métodos associados às transições e aos estados definem a forma como os recursos do sistema controlado são usados e modificados.

5.6.2 Objectos activos e passivos

Um *objecto activo* está continuamente a executar (tem o seu próprio fio de execução) e é geralmente autónomo, o que significa que exibe algum comportamento, sem a necessidade do comando dum outro objecto. Os objectos desta categoria contrastam com os *objectos passivos*, que não tomam qualquer iniciativa de iniciar comunicação e que, portanto, só actuam a pedido de outros objectos. Assim, os objectos activos servem de base para o controlo (ou supervisão) de todo o sistema, enquanto que os objectos passivos, sempre que solicitados, fornecem serviços àqueles. Tipicamente os objectos-função são activos e os objectos-informação são passivos.

A definição de objectos activos e passivos aqui enunciada é semelhante à que é proposta na metodologia HOOD [Robinson, 1992, pág. 51].

Em UML, um objecto activo ou uma classe activa são representados pelos seus símbolos habituais (um rectângulo), mas com o bordo mais carregado. A fig. 5.13 apresenta um exemplo dum objecto activo e duma classe activa (comparar com a fig. 3.4 em que se apresentam um objecto passivo e uma classe passiva).

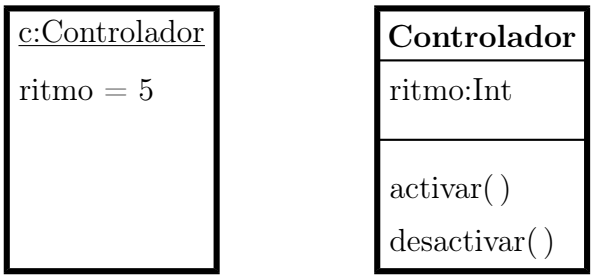


Figura 5.13: Notação gráfica para objectos activos e classes activas.

Repare-se que esta classificação dos objectos é ortogonal, já que pode ter-se, por exemplo, um objecto passivo, que tenha um comportamento simples, e outro objecto passivo, cujo comportamento seja reactivo.

5.6.3 Regras para a utilização de state-charts

Quando um objecto é criado, o respectivo state-chart, caso exista, não se inicia imediatamente num dado estado, mas antes uma transição inicial é disparada. Esta transição pode representar um novo fio de execução e constitui uma das características que distingue um objecto reactivo dum objecto simples. Só pode haver, como é óbvio, um ponto inicial (global) para cada state-chart.

Um objecto que tenha o seu comportamento descrito por um state-chart, tem o seu estado global (ou estendido) definido pela combinação do estado do state-chart (em que se encontra em cada instante) com o valor dos atributos que lhe estão associados. Os estados do state-chart representam uma vista abstracta do objecto e focam nos aspectos qualitativos do comportamento, enquanto que as variáveis (mais concretamente, os seus valores) se relacionam com os aspectos quantitativos. Segundo esta perspectiva, uma mudança no valor duma variável não implica necessariamente uma mudança de estado.

Os nomes a dar aos estados devem ser o mais significativos possível, reflectindo o estado (modo) em que o sistema se encontra. Daí que uma escolha apropriada desses nomes se revele fun-

damental para uma maior legibilidade dos state-charts. Em entender do autor, nomes como "Calculando" ou "Em Cálculo" são preferíveis a "Calcular" ou "Cálculo", por os primeiros terem associada uma noção de tempo, ao contrário dos últimos que têm uma essência mais efémera.

Repare-se que, nos métodos estruturados, o uso de máquinas de estado é frequente, porém não há uma definição clara de quais as partes do sistema que exibem esse comportamento, regulado por estados. Nas metodologias orientadas ao objecto, as máquinas de estado são unicamente definidas em classes, cujas instâncias as executam.

Apesar dos objectos activos estarem continuamente a supervisionar fluxos de informação (sejam contínuos ou discretos) ou outros objectos-entidade susceptíveis de se alterarem, podem ter igualmente como entradas interacções e eventos. Trata-se do único tipo de objecto que tem capacidade para uma actuação independente, na medida em que os objectos passivos desempenham apenas os serviços que lhe estão consignados, quando estimulados exteriormente (por um evento ou por uma interacção). Se um objecto tem de monitorizar continuamente o valor duma informação, então o seu comportamento deve ser especificado de modo activo. Um exemplo característico deste tipo de comportamento é o dum objecto que gera um alarme (evento), sempre que o nível dum dado gás tóxico se encontrar acima dum valor pré-fixado. Usualmente, os objectos-função podem apresentar um comportamento activo.

Um evento será ignorado, caso ocorra quando a máquina está num dado estado que não é sensível a esse evento. Uma situação especial nas transições verifica-se quando não se indica qualquer evento (transição automática). Em UML, assume-se que, assim que a actividade do estado a montante terminar, ocorre implicitamente um evento "fim de actividade" (*completion event*) que habilita o disparo da transição automática. Contudo, caso uma condição esteja associada a essa transição automática, esta só dispara se, nesse ciclo, aquela for avaliada como verdadeira. Em caso contrário, o evento "fim de actividade" é ignorado e só tornará a ocorrer, para o estado actual, assim que ele for abandonado (por outra transição) e retomado.

A fig. 5.14 ilustra 3 situações distintas de modelação, cuja semântica de disparo das transições é distinta e que importa aqui realçar, devido à sua importância para a especificação de sistemas reactivos.

Na primeira situação, após terminar a actividade de E1, a transição automática fica imediatamente habilitada. Se a condição cond1 for avaliada como verdadeira, a transição dispara, passando a máquina a estar no estado E2. Se, porém, cond1 for falsa, o evento "fim de actividade" é ignorado e o estado E1 terá de ser abandonado por outra transição com origem no estado E1 (ou em qualquer um dos seus super-estados). Neste caso, se não existir nenhuma transição além da retratada na fig. 5.14(a), a máquina de estados ficará eternamente bloqueada no estado E1, situação normalmente não pretendida.

O segundo caso, representado na fig. 5.14(b), mostra que a expressão "when cond3" representa um evento transição (*change event*) que é gerado implicitamente, assim que a condição booleana indicada (no caso cond3) for avaliada como verdadeira, devido a alterações nos valores dos seus constituintes. Assim sendo, a transição dispara, mal a condição cond3 seja avaliada como verdadeira, independentemente da actividade a3 ter ou não terminado. Pode ver-se um evento transição como um teste em contínuo à expressão booleana indicada (pág. 94).

A terceira e última situação, ilustrada na fig. 5.14(c), mostra que a máquina de estados muda do estado E5 para o estado E6, apenas quando a actividade a5 concluir. Depois desta transição de estado (i.e. depois da actividade estar garantidamente concluída), a máquina passa a estar sensível à condição cond6. A diferença relativamente à fig. 5.14(b) consiste no facto de, nesse

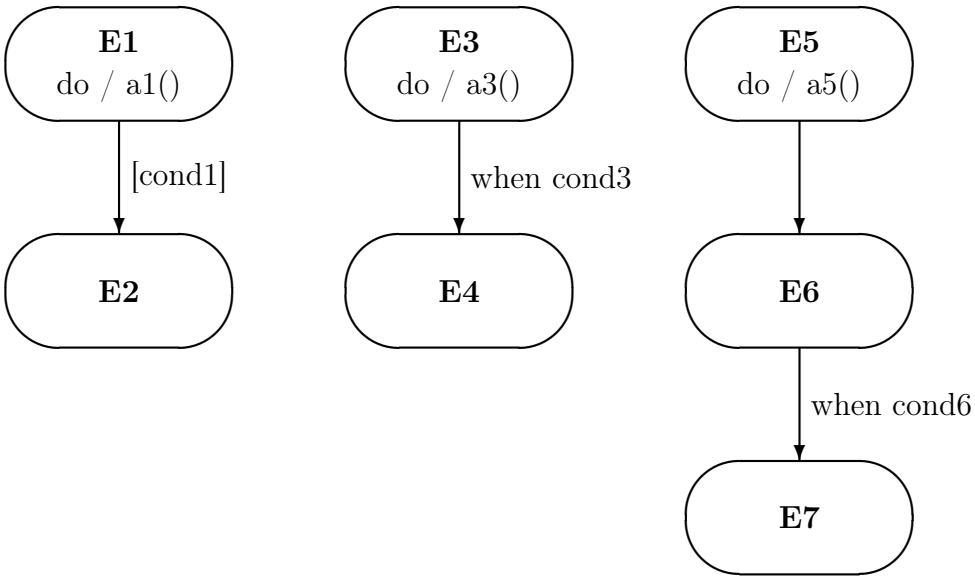


Figura 5.14: Situações de modelação com tipos distintos de etiquetas nas transições.

caso, a actividade poder ser interrompida, o que não sucede nesta situação, pois a primeira transição dispara imediatamente após a actividade estar concluída.

A fig. 5.15 mostra 2 situações de modelação, em que se recorreu a eventos temporais nas etiquetas das transições. Se um estado tem associada uma transição de saída com um evento temporal, é necessário sempre que esse estado for activado iniciar um temporizador que arranca com o valor T, indicado como parâmetro do evento tm(T).

Note-se que não faz sentido associar duas ou mais transições de saída com eventos temporais, pois apenas a que tiver associado o tempo menor disparará. Igualmente indesejável, por tornar o modelo não determinístico, é a existência de duas transições com eventos temporais iguais.

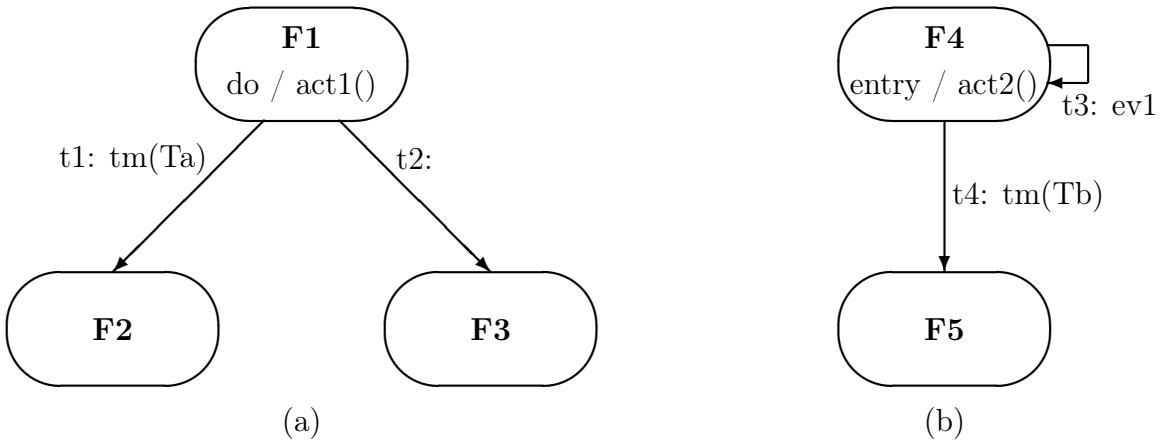


Figura 5.15: Situações de modelação com recurso a eventos temporais nas etiquetas das transições.

O primeiro caso, ilustrado na fig. 5.15(a), mostra que a transição t1 dispara, se o tempo Ta esgotar antes de terminar a actividade act1. Se a actividade se concluir, antes do tempo Ta se

esgotar, será a transição t2 a disparar. Neste caso, há a garantia de que uma das 2 transições disparará.

No segundo caso, o temporizador com o tempo Tb deve ser reiniciado, sempre que o evento ev1 for activado. Repare-se que este evento só dispara se ocorrer antes do tempo Tb se esgotar e implica a execução da actividade *entry* act2 (i.e. a máquina sai de F4 e volta a entrar em F4).

5.6.4 Modelo temporal dos state-charts

A definição do modelo do tempo associado aos state-charts é crucial para que a respectiva semântica seja clara, de modo a determinar se, por exemplo, alterações produzidas num dado passo de execução (eventos gerados ou atributos modificados) devem reflectir-se no mesmo passo (i.e. imediatamente) ou apenas no seguinte [Harel e Naamad, 1996].

O primeiro modelo temporal é designado por *assíncrono* e considera que o sistema reage assim que uma alteração externa ocorrer, o que permite que vários passos possam suceder no mesmo instante temporal. O segundo modelo do tempo é designado por *síncrono* e assume que o sistema executa um único passo em cada unidade temporal, reagindo a todas as alterações externas que ocorreram desde o ciclo anterior. Este modelo temporal pressupõe a existência dum sinal regulador do avanço do sistema, habitualmente designado de *relógio*. Considere-se o state-chart da fig. 5.16 como referência para ilustrar as diferenças desses dois modelos do tempo e atente-se que, inicialmente, a máquina de estados está no estado {A,C}.

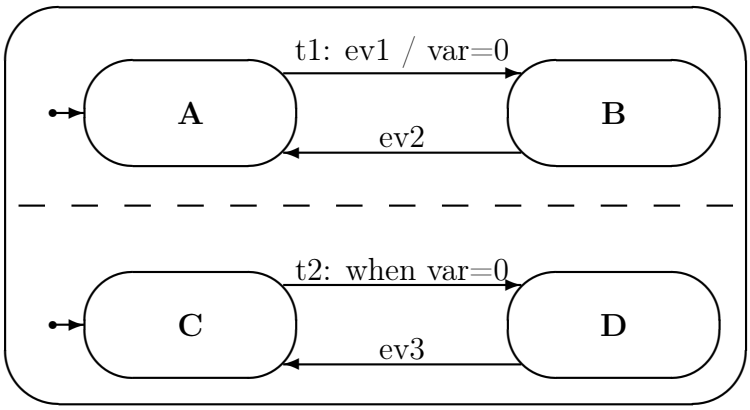


Figura 5.16: State-chart para exemplificar as diferenças entre os modelos temporais síncrono e assíncrono.

Comece-se por considerar o modelo do tempo *síncrono*. Assim que surgir o evento ev1, a transição t1 está habilitada a disparar no ciclo seguinte. O disparo dessa transição dá-se assim que surgir um pulso no sinal de relógio e tem por consequência atribuir à variável var o valor 0 e activar o estado B, desactivando simultaneamente o estado A. A máquina passa a estar no estado {B,C}. No ciclo seguinte (i.e. no pulso seguinte do sinal de relógio), a transição t2 está habilitada a disparar, uma vez que a variável var passou a ter valor 0. Quando a transição t2 dispara, a máquina passa a estar no estado {B,D}. A máquina continuará a executar, normalmente, mas para esta explicação o resto do seu comportamento não é relevante.

Considere-se agora a execução do mesmo state-chart, mas usando a semântica imposta pelo modelo do tempo *assíncrono*, em que não existe associado o conceito de relógio, como regulador

da passagem do tempo. Mal surja o evento ev1, a transição t1 dispara, o valor 0 é atribuído à variável var e a máquina passa, momentaneamente a estar no estado {B,C}. O uso do termo "*momentaneamente*" é de salientar, pois, imediatamente ao disparo de t1, deve disparar-se t2, dado ser válida a condição que lhe está associada ([var=0]). Pode afirmar-se que a alto nível, a máquina passa directamente do estado {A,C} para o estado {B,D}. Nessa altura, a máquina atinge um estado estável, continuando a máquina a executar (neste caso, à espera dos eventos ev2 ou ev3).

Em MIDAS, a escolha recaiu sobre o modelo síncrono em detrimento do modelo assíncrono, pois aquele permite, com mais facilidade, gerar (manual ou automaticamente) código para síntese, facilita o processo de simulação e adapta-se melhor na especificação de sistemas digitais síncronos. Esta última característica é muito importante, pois, actualmente, a implementação de sistemas digitais síncronos apresenta-se ainda como a solução mais simples e com maior probabilidade de funcionamento correcto, apesar das desvantagens que lhe estão associadas, nomeadamente a dificuldade em distribuir o sinal global de relógio [De Micheli, 1994, pág. 6].

Deve notar-se que a linguagem VHDL, que é usada na metodologia MIDAS para síntese das componentes a implementar em hardware, tem um modelo do tempo que segue igualmente o princípio que as alterações produzidas num dado ciclo ("*delta cycle*" na terminologia VHDL) só serem visíveis no ciclo seguinte. Assim sendo, o código VHDL a gerar deverá ser escrito ao nível da transferência de registos (RTL), usando-se um sinal global —o relógio—, cujas transições de valor, num dos bordos (ascendente ou descendente), indicam quando um novo ciclo de execução se produz.

Caso a escolha tivesse recaído sobre o modelo temporal assíncrono, o código VHDL poderia, por exemplo, ter sido escrito ao nível comportamental. Neste estilo de codificação, o sistema, especificado por um conjunto de processos, reage imediatamente a qualquer alteração das entradas a que é sensível no estado em que estiver activo.

5.6.5 Herança de state-charts

Ao definir-se o comportamento duma classe de objectos, através dum state-chart, emergem alguns problemas relacionados com a herança. Uma solução para este problema poderia ser ignorar completamente o state-chart da superclasse e desenhar de raiz um novo state-chart (possivelmente com uma estrutura completamente diferenciada) para a subclasse. Esta solução não será, contudo, considerada, pois é contrária ao espírito da modelação orientada ao objecto.

Assim, o diagrama de state-charts, definido numa classe para descrever o comportamento dos objectos, deve ser herdado pelas subclasses, mas para tal têm de ser cumpridas algumas regras. Essas regras terão de ser, de alguma forma, semelhantes àquelas que normalmente se seguem quando a herança entre classes se faz directamente no código, através dos atributos e das operações. Nesta subsecção, pretende indicar-se algumas dessas regras, referindo os problemas que lhe são inerentes.

Dada a natureza das modificações que um state-chart pode sofrer em subclasses, não foi ainda possível encontrar uma forma, simples e natural, de explicitar incrementalmente essas modificações. O método de análise OSA apresenta uma notação em que um *state-net* (o nome dado aos modelos de estados) pode ser especificado incrementalmente, tendo em consideração o state-net da superclasse [Embley et al., 1992, pág. 90–1]. Esta notação reduz as possibilidades de se cometerem erros e incoerências, já que se procede a uma especificação diferencial, o que faz

realçar as diferenças entre o comportamento genérico e o especializado. No entanto, a notação só se revela eficaz quando as alterações são reduzidas e localizadas, uma vez que força uma disposição espacial dos elementos de modelação, para permitir a construção do modelo completo a partir dos diagramas das superclasses, pelo que não pode ser adoptada genericamente.

Dadas as limitações acima indicadas, em Midas, propõe-se que o state-chart associado a uma subclasse seja, graficamente, um diagrama completo. Não se pode, portanto, usar a facilidade de expressão adoptada, nas linguagens de programação orientadas ao objecto, em que numa subclasse se indicam apenas as operações que foram acrescentadas ou modificadas, sendo todas as outras implicitamente herdadas sem qualquer alteração.

Quando o state-chart duma subclasse é um diagrama completo, existem algumas propostas que pretendem fazer salientar as alterações introduzidas na subclasse, usando, para os elementos herdados, símbolos distintos daqueles introduzidos pela própria classe. Uma das propostas usa símbolos a tracejado para os elementos herdados e símbolos a traço normal para novos elementos [Weber e Metz, 1998], enquanto que outra recorre a símbolos em tons de cinzento para os elementos herdados e símbolos a cor preta para novos elementos [Selic et al., 1994, pág. 268]. Porém, em qualquer dos casos, é sempre necessário desenhar o diagrama completo, pelo que, se for feita alguma modificação no state-chart duma superclasse, ela, infelizmente, não se reproduz automaticamente nos state-charts das suas subclasses, o que obriga a reproduzir essa alteração em todas estas. Deste facto podem resultar, com alguma facilidade, incoerências entre as várias classes, o que é um efeito indesejável que deve ser evitado.

Outra alternativa é transformar os state-charts em código (através de pré-condições, operações e pós-condições), o que permite assim herdar implicitamente os state-charts via código [Oblog, 1997a]. A principal desvantagem desta solução é que se perde a visão gráfica dos state-charts, mas, por contrapartida, ganha-se a possibilidade de especificar um state-chart de forma diferencial. Em Midas, optou-se por manter, na fase de análise, o carácter gráfico dos state-charts, podendo, no entanto, nas fases mais finais do projecto, transformar-se essa informação num formato textual (em Oblog).

Para mais facilmente se compreender como deve ser interpretada a herança entre state-charts, deve perspectivar-se a sua utilização admitindo que se pode transformar o state-chart associado a uma classe em atributos e operações dessa classe (discussão mais pormenorizada no cap. 6). O primeiro passo na transformação do state-chart em propriedades da classe consiste em criar um atributo varEstado, cujo valor, em cada instante, representa o estado actual do objecto. Cada estado dá origem a uma operação da classe, devendo igualmente todos os eventos, actividades e acções ter uma operação disponível na classe. Tomando-se esta perspectiva, as considerações que foram tecidas na secção 5.5 relativamente ao mecanismo de herança devem, por uma questão de coerência, ser igualmente seguidas relativamente aos state-charts.

Antes de apresentar as regras, convém esclarecer o que significa um objecto da classe A ser também um objecto da classe (mais genérica) B. Na maioria das abordagens à herança, a relação "é_um" entre as classes A e B impõe como requisito mínimo a conformidade das interfaces. Isto significa que é possível colocar uma instância de B, em qualquer local onde uma instância de A também pode ser colocada, desde que a interface de B (aquilo que B pode executar) seja consistente com a interface de A.

Porém, esta exigência não implica que haja conformidade comportamental, entre as classes A e B. Apenas se garante que podem substituir-se instâncias de A por instâncias de B, sem provocar incompatibilidades, mas, concomitantemente, sem qualquer certeza que a forma como os objectos B reagem é compatível com aquela que os objectos A exibem. De facto, a resposta

dum objecto B a um evento ou operação pode ser totalmente diferente daquela que um objecto A teria. Em termos práticos, garantir completamente a compatibilidade comportamental entre duas classes é tecnicamente muito difícil [Harel e Gery, 1997].

No entanto, na maioria das situações, não se exige que a relação hierárquica entre uma classe e a sua superclasse signifique que B faz o mesmo que A e da mesma forma. É normalmente apenas pretendido que uma subclasse B responda aos mesmos estímulos a que a classe A responde, mas podendo fazê-lo de modo completamente distinto. Esta postura pressupõe que a herança é introduzida com o propósito principal de facilitar a reutilização.

No quadro 5.1, apresentam-se 8 regras que se podem aplicar a um state-chart especificado numa classe para o adaptar ao comportamento duma sua subclasse. Foi feita uma análise às propostas de herança sugeridas por diversos autores, o que permitiu seleccionar as regras mais relevantes para o tipo de sistemas considerado neste trabalho. Todas estas propostas apresentam o processo de herança segundo a perspectiva da especialização (das superclasses para as subclasses), embora seja também possível realizá-lo por generalização (das subclasses para as superclasses).

Regra	Weber & Metz	OSA	Real Time UML	Harel	ROOM	OOA	OMT
h1. Conservação do state-chart	sim	sim	sim	sim	sim	sim	sim
h2. Redefinição das actividades e acções dum estado	sim	sim	sim	sim	sim	não	n.d.
h3. Adição de transições e estados	sim	sim	sim	sim	sim	sim	restrito
h4. Alteração do estado destino duma transição	sim	não	sim	sim	sim	não	não
h5. Remoção de transições	parcial	não	não	não	sim	não	não
h6. Especialização de etiquetas de transições	sim	sim	sim	sim	sim	sim	n.d.
h7. Remoção de estados	não	não	não	não	sim	não	não
h8. Alteração do estado origem duma transição	não	não	não	não	sim	não	não

Tabela 5.1: Regras suportadas nas seguintes propostas: Weber & Metz [Weber e Metz, 1998], OSA [Embley et al., 1992, pág. 88–91], REAL-TIME UML [Douglass, 1998, pág. 174–6], Harel [Harel e Gery, 1997], ROOM [Selic et al., 1994, cap. 9], OOA [Shlaer e Mellor, 1992, pág. 57–60] e OMT [Rumbaugh et al., 1991, pág. 111].

Das 8 regras consideradas, sugere-se, em MIDAS, a adopção das 6 primeiras (h1 a h6), não se incentivando o uso das 2 últimas (h7 e h8). Na secção 6.4, apresentam-se exemplos concretos da utilização de cada uma das 6 regras adoptadas e a sua relação com o código OBLOG.

[h1] Conservação do state-chart

Nas situações em que, na subclasse, são adicionados atributos e operações que não têm qualquer dependência do estado do objecto, deve o state-chart da superclasse ser usado, sem qualquer modificação, para descrever o comportamento dos objectos da subclasse. Esta situação implica que os atributos e as operações adicionados podem ser usados, em qualquer instante, durante a existência do objecto.

[h2] Redefinição das actividades e acções dum estado

No state-chart da subclasse, deve ser possível redefinir (leia-se, especializar ou acrescentar) as actividades e as acções (do tipo *entry* e *exit*) associadas a um estado.

É possível, em Real-Time UML, remover, sem qualquer restrição, acções e actividades dum estado. De acordo com o pressuposto inicial, esta possibilidade só deve ser contemplada se a operação associada à acção ou actividade removida, continuar a ser referida no state-chart. Se tal sucede, essa operação continua disponível na subclasse. Caso se removam, do state-chart, todas as referências a uma dada operação, então, por herança, a subclasse passa, a poder usar uma operação, cujo acesso não está previsto, podendo assim criar-se uma incoerência.

[h3] Adição de estados e transições

Uma subclasse pode incorporar estados e transições ao state-chart definido na superclasse. Esta hipótese está intimamente relacionada com o uso do mecanismo de herança para acrescentar, na subclasse, atributos e operações. Acrescentar um novo estado, obriga a acrescentar novas transições ligadas a estados herdados e, eventualmente, a adicionar eventos (herdados ou definidos na subclasse) a essas novas transições.

Nesta linha de raciocínio, acrescentar novas transições a estados herdados é perfeitamente válido, uma vez que se trata dum caso particular desta regra. Contudo, deve garantir-se que as guardas associadas a todas as transições originárias dum dado estado, são disjuntas entre si, caso se pretenda um state-chart determinístico. Esta pretensão pode obrigar a redefinir as guardas associadas às transições herdadas.

A palavra adição desta regra deve ser interpretada duma forma lata. As propostas de Harel e Gery e de Douglass incluem as seguintes alterações a um estado, cuja consequência prática e visível é um state-chart com mais estados:

- Decompor um estado em sub-estados concorrentes (componentes ortogonais).
- Adicionar sub-estados a um estado concorrente.
- Adicionar componentes ortogonais a um estado sequencial.

A proposta de OMT, possivelmente por ser a mais antiga, é a mais restritiva, uma vez que não permite a introdução directa de novos estados e transições, no state-chart da superclasse, porque este deve ser uma projecção do state-chart da subclasse, ou, visto duma outra perspectiva, este deve ser um refinamento daquele. Assim, qualquer estado do state-chart da superclasse pode ser especializado[11] ou dividido em partes concorrentes.

[11] Apesar de ter sido utilizada a palavra "generalizado" no livro [Rumbaugh et al., 1991], a troca de correspondência, via correio electrónico, permitiu esclarecer, junto do primeiro autor, que a palavra mais adequada seria "especializado".

[h4] Alteração do estado destino duma transição

A modificação do estado destino duma transição deve ser aceite, desde que continue a ser possível atingir o anterior estado destino por uma outra transição. A verificação desta condição garante que as actividades e acções do estado destino continuam a ser necessárias no state-chart da subclasse.

A utilização desta regra é, normalmente, necessária quando se introduz, por exemplo, um novo estado no meio de dois estados herdados. Neste caso, o novo estado passa a ser o destino da transição que na superclasse ligava os dois estados herdados, adicionando-se uma transição nova entre o novo estado e o estado onde terminava a transição alterada.

[h5] Remoção de transições

A remoção duma transição do state-chart da superclasse não provoca qualquer incoerência, em termos das operações a que a subclasse tem acesso, desde que, pelo menos, uma transição no state-chart da subclasse mantenha o evento que está associado à transição eliminada. Se todas as transições associadas a um dado evento forem eliminadas, a subclasse continua, através do mecanismo de herança, a poder usar esse evento, criando-se assim uma incoerência. Neste sentido, a eliminação de transições automáticas (incondicionais) não produz qualquer problema e pode assim ser realizada. A remoção duma transição pode implicar a redefinição das guardas associadas a outras transições que partem do mesmo estado origem.

É possível eliminar implicitamente uma transição, redefinindo a condição que lhe está associada com o valor falso (através da regra h6). Com este artifício, a transição continua a estar representada no state-chart, mas nunca é disparada, pois a condição com o valor falso não o permite.

[h6] Especialização de etiquetas de transições

As transições estão etiquetadas com inscrições que seguem o formato *Evento[Guarda]/Acção*. A especialização das etiquetas pressupõe que na subclasse se pode alterar (i.e. acrescentar, eliminar ou modificar) o evento, a condição e as acções. Como se referiu na regra h5, quando se altera a condição duma transição para a constante falso, está-se implicitamente a eliminar essa transição.

[h7] Remoção de estados

A remoção dum estado não deve ser permitida, pois a subclasse deixa de ter acesso a um estado que está definido na superclasse. Se realmente for eliminado um estado da superclasse, está a criar-se uma situação de incoerência, já que a operação associada a esse estado está definida na superclasse e, portanto, disponível, por herança na subclasse.

Uma forma de tornear esta impossibilidade consiste em eliminar todas as transições que terminam nesse estado, colocando a falso as respectivas guardas (através da regra h6). Como consequência, está implicitamente a eliminar-se o estado pretendido. Embora o estado continue a estar representado no state-chart, na prática, ele foi eliminado pois está isolado do resto do state-chart.

[h8] Alteração do estado origem duma transição

Normalmente, não é permitido alterar o estado origem duma transição. Apesar desta regra não ser usualmente autorizada, não parece haver nenhuma razão para tal, segundo os pressupostos que se fizeram inicialmente.

5.7 Diagrama de sequência

Os diagramas de casos de uso, para serem uma técnica de descrição útil e poderosa, não podem ser considerados isoladamente, já que contêm pouca informação sobre a funcionalidade do sistema. Contudo, a sua utilização como estrutura referencial para outros diagramas (nomeadamente, diagramas de objectos e de estados) que descrevem, em pormenor, a sequência de acções do sistema é extremamente pertinente.

Assim, a utilização de diagramas de sequência, na metodologia Midas, enquadra-se nesta perspectiva. Retomando a fig. 4.1, constata-se ainda que os diagramas de sequência são também usados para validar o processo de simulação, cujos resultados são gerados no mesmo tipo de diagrama para facilitar essa tarefa de comparação.

5.8 Resumo final

Neste capítulo, descreveu-se a forma como a fase de análise deve ser seguida no âmbito da metodologia Midas, apresentando os diversos passos e a sua relação.

A fase de análise, em Midas, inicia-se com a construção do diagrama de contexto que mostra o sistema, representado como uma entidade única, rodeado por todos os actores que com ele interactuam. O diagrama de contexto define a fronteira entre o sistema e o seu ambiente, o que é útil para demarcar o limite do sistema a desenvolver. De seguida, recorre-se a casos de uso para indicar as funcionalidades que o sistema deve disponibilizar aos seus utilizadores. A construção do diagrama de objectos é a tarefa seguinte e tem como ponto de partida os casos de uso. Foram indicadas algumas recomendações para esta transição, que se baseia na categorização dos objectos (objectos-entidade, objectos-função e objectos-interface). Outros assuntos foram também referidos, nomeadamente, a composição de objectos, os objectos repetidos, a criação e a eliminação dinâmicas de objectos, a introdução de questões temporais e a escolha de nomes apropriados.

De seguida, constrói-se o diagrama de classes que é visto, em Midas, como um repositório de especificações pré-definidas de objectos, passíveis de serem reaproveitadas para qualquer aplicação. O uso de relações de herança é fortemente incentivado, em contraste com as relações de agregação e composição. Em Midas, sugere-se estruturar as classes em árvore, sendo as folhas da estrutura classes concretas e qualquer superclasse uma classe abstracta. O uso de herança não estrita, múltipla e repetida não é aconselhado, mas também não é expressamente interdito.

Em relação à especificação do comportamento dum sistema/objecto, o uso de state-charts mostra-se relevante para objectos reactivos. Foram apresentadas algumas regras que devem ser cumpridas, de forma a assegurar uma adequada especificação dos sistemas. Para a semântica dos state-charts, optou-se pela adopção do modelo temporal síncrono, devido à facilidade

em gerar código, em simular e em implementar o sistema. A herança entre state-charts de classes relacionadas foi também abordada, tendo sido analisadas 8 regras propostas noutras metodologias, das quais se escolheram 6.

Capítulo 6

A Representação Unificada OBLOG

Quem faz o que pode faz o que deve.

Sumário

Neste capítulo, aborda-se a utilização da linguagem OBLOG *como representação unificada para sistemas embebidos. São apresentadas algumas recomendações genéricas que indicam a forma de transformar os diagramas* UML *(diagramas de classes, objectos e state-charts) no código* OBLOG*. É dado especial ênfase à transformação de state-charts para código, por ser essa a parte mais complexa de todo o processo de codificação, devido aos mecanismos de modelação intrínsecos ao meta-modelo state-chart. A apresentação da tradução é feita com recurso a uma série de casos típicos de modelação, envolvendo transições iniciais, acções, actividades, transições a vários níveis de profundidade, transições automáticas, transições condicionais, transições de grupo, conectores história, e state-charts com hierarquia. O capítulo termina com uma análise sobre a forma como as relações hierárquicas entre classes se reflectem no código* OBLOG*, tendo em conta as 6 regras de herança entre state-charts que podem aplicar-se no âmbito da metodologia* MIDAS*.*

Índice

6.1 Considerações iniciais

Qualquer metodologia que contemple a fase de implementação deve criar uma especificação do sistema, escrita numa dada linguagem de programação (ou num conjunto delas). Este passo levanta algumas dificuldades, pois é necessário gerar, a partir dum conjunto de diagramas, uma especificação, escrita em código textual.

No caso da metodologia MIDAS, a geração do código final para implementação é feita a partir duma especificação intermédia, escrita em OBLOG, uma linguagem de modelação (e não de programação) orientada ao objecto, independente da plataforma alvo de implementação. A sintaxe e a semântica de OBLOG não são apresentadas neste trabalho, por estarem devidamente documentadas [Oblog, 1997a] [Oblog, 1997b] [Oblog, 1997c].

Assim, antes de passar à fase de concepção, deve criar-se uma especificação OBLOG (*repositório* na terminologia OBLOG) a partir da informação que consta dos diagramas de objectos, diagramas de classes e state-charts, podendo, adicionalmente, utilizar-se os diagramas de sequência como uma fonte de informação alternativa. Nas fases de concepção e implementação, o repositório OBLOG é manipulado, de forma a obter especificações escritas em C e em VHDL para as partes a implementar, respectivamente, em software e em hardware. A obtenção das especificações em C e VHDL é feita de forma automática, dado que foram construídos compiladores (*scripts*) que traduzem OBLOG nessas 2 linguagens [Pereira e Paredes, 2000]. A escolha das linguagens C e VHDL resultou da sua enorme popularidade para implementar funcionalidades em software e em hardware, respectivamente, mas, através do mecanismo de scripts OBLOG, é possível abrir o leque de opções a outras linguagens (por exemplo, JAVA, C++, VERILOG).

Nesta secção, apresentam-se algumas recomendações que permitem, duma forma manual e iterativa, "traduzir" os diagramas atrás mencionados num repositório OBLOG, que descreve uma comunidade de objectos concorrentes que interagem entre si.

Apesar de, na fase de análise, não se misturarem, no mesmo diagrama, classes e objectos, não é necessário evitar essa coexistência ao nível do repositório OBLOG. Esta linguagem não disponibiliza nenhum construtor que permita criar directamente, ao nível da especificação, instâncias duma dada classe. Essa possibilidade existe apenas ao nível da simulação (execução do modelo). Contudo, a linguagem inclui o conceito de objecto (construtor `object`), como sendo uma classe que tem uma e uma só instância. Neste sentido, pode usar-se o construtor `object` da linguagem OBLOG para especificar todos os objectos que compõem o sistema e o construtor `class` para especificar as classes a que pertencem esses objectos. Entre as classes podem indicar-se relações de herança e entre os objectos é possível explicitar associações que indicam que há, entre eles, comunicação (troca de informação).

A fig. 6.1 mostra a forma como os diagramas de classes e de objectos podem ser descritos em OBLOG. A linha a tracejado representa uma divisão conceptual: acima da linha a informação é proveniente do diagrama de classes e abaixo é originária do diagrama de objectos. As setas que cruzam a linha respeitam à relação objecto/classe. Apesar do constructor `object` representar uma classe ≪singleton≫ (e não um objecto), assume-se implicitamente que a respectiva instância faz parte do modelo dos objectos. Se forem necessárias 2 instâncias duma dada classe (por exemplo, ClasseD), em OBLOG têm de ser criadas 2 subclasses de ClasseD (objD1 e objD2), que acabam simplesmente por representar as 2 instâncias desejadas. Deste modo, parece aceitável que a fig. 6.1 represente simultaneamente os diagramas de classes e de objectos, apesar de só estarem representadas classes. Assume-se, para não sobrecarregar a figura, que por baixo de

cada classe ≪singleton≫ está associada a respectiva instância.

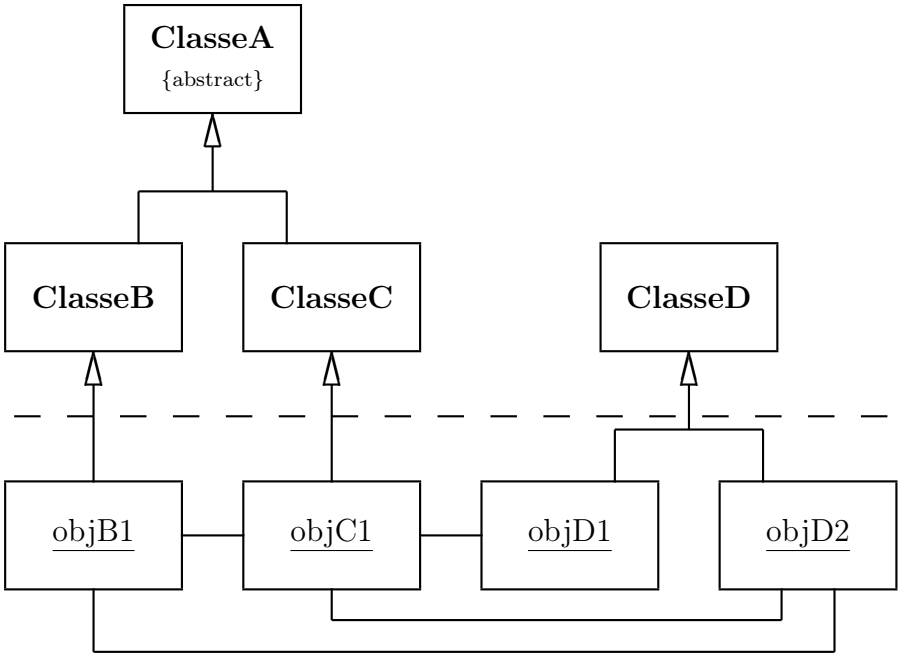


Figura 6.1: A forma como as classes e os objectos podem ser organizados em Oblog.

Para cada classe do diagrama de classes, cria-se uma especificação de classe Oblog, usando-se a palavra-chave `abstract`, nas especificações de classes que sejam abstractas (que não possam ter directamente instâncias). É necessário ter em atenção que, em Oblog, existem atributos e operações de classes e de objectos. Os atributos de classe referem-se a valores com relevância para todas as instâncias dessa classe (por exemplo, o número de instâncias existentes), ao contrário dos atributos de objectos (também denominados variáveis de instância), cujo âmbito se limita ao objecto correspondente. As operações de classe respeitam a acções relevantes a toda a classe (o exemplo típico é a criação de instâncias dessa classe que implica a actualização do atributo de classe que retém o número de instâncias existentes), enquanto que as operações de objectos se repercutem apenas no contexto do objecto particular em que a operação é invocada.

Para cada objecto presente no diagrama de objectos, cria-se uma especificação de objecto Oblog. Para os objectos não é preciso criar na secção de declaração atributos e operações pois, em princípio, estes foram definidos na classe respectiva. É recomendado usar a palavra-chave `active`, nas especificações de objectos e classes que sejam activos, conforme definição dada na secção 5.6. Esta informação é relevante para as fases de desenvolvimento posteriores, pois permite identificar rapidamente os fios de execução do sistema.

Para um objecto que tenha associado um state-chart, é necessário fazer reflectir essa informação no respectivo código Oblog. A transformação de state-charts em código consiste provavelmente na parte mais complexa do processo de construção do repositório, devido aos vários mecanismos de modelação intrínsecos ao meta-modelo state-chart. Há várias maneiras de implementar uma máquina de estados em software, sendo a mais comum conseguida com recurso a uma variável dum tipo enumerado (designada varEstado) que serve como variável de selecção num comando `switch`. Cada cláusula `case` implementa um dado estado do diagrama.

Seguidamente, é listado, em C/C++, o código respeitante ao state-chart apresentado na fig. 6.2, segundo esta abordagem de tradução, proposta por diversos autores [Douglass, 1998, pág. 192]

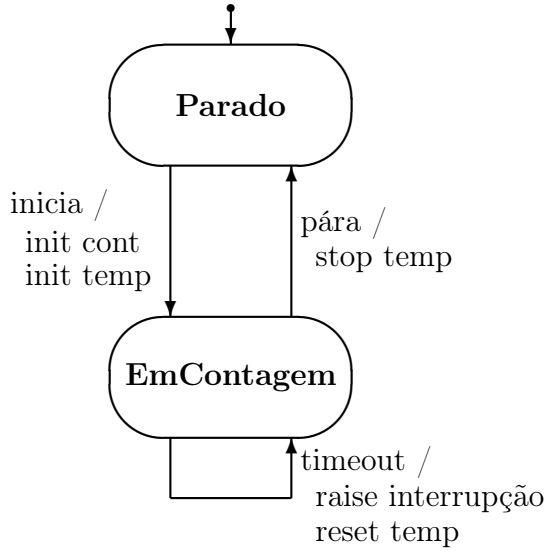


Figura 6.2: Um state-chart com 2 estados.

[Booch et al., 1999, pág. 338].

```
 1:  void StateChart(TEstado &varEstado, TEvento ev) {
 2:    switch(varEstado) {
 3:      case Parado:
 4:        switch (ev) {
 5:          case INICIA:
 6:            temp.countValue = msg.cmd;
 7:            temp.start();
 8:            varEstado = EmContagem;
 9:            break;
10:          default:
11:            // não faz nada
12:            break;
13:          };
14:        break;
15:      case EmContagem:
16:        switch (ev) {
17:          case TIMEOUT:
18:            sw_interrupt(xx);
19:            temp.start();
20:            break;
21:          case PARA:
22:            temp.stop();
23:            varEstado = Pronto;
24:            break;
25:          default:
26:            // não faz nada
27:            break;
28:          };
29:        break;
30:      default:
31:        cout << "Estado ilegal" << endl;
32:        break;
33:    };
34:  };
```

O código apresentado coloca os estados num nível de escolha superior [`switch(varEstado)`] e, para cada estado, é criado um novo nível de escolha baseado nos eventos ou mensagens [`switch(ev)`]. Alternativamente, poderia inverter-se esta hierarquização (eventos primeiro e estados depois), mas dessa inversão não resultariam mudanças significativas no tipo de tradução a efectuar.

Esta abordagem tem a vantagem de ser conceptualmente simples, mas torna-se intratável quando aplicada a máquinas de estado relativamente complexas (especialmente, modelos descritos de forma hierárquica). De facto, esta abordagem obriga a considerar a máquina de estados plana, o que, em alguns casos, produz um aumento exponencial no número de estados e transições [Selic et al., 1994, pág. 340]. Por este motivo, não será seguida, neste trabalho, esta hipótese para a tradução de state-charts em código.

No âmbito da modelação por objectos, vários autores apresentaram algumas propostas que sugerem a forma como deve ser gerado código, tendo em consideração as máquinas de estado que descrevem o comportamento interno dos objectos. Entre essas propostas, salientem-se os padrões de concepção (*design patterns*) State [Gamma et al., 1995] e State Table [Douglass, 1998, pág. 287–301] e a solução preconizada por [Shlaer e Mellor, 1992].

Estas soluções foram idealizadas para sistemas software, onde princípios como a reutilização, a extensibilidade e a simplicidade dos modelos e do respectivo código são mais críticos do que propriamente o desempenho e a utilização de memória. Contudo, para sistema embebidos, estes dois últimos conceitos revelam-se importantíssimos, pois há requisitos temporais que têm de ser cumpridos e, regra geral, existem fortes limitações no espaço de memória disponível. Assim sendo, preconizou-se uma solução distinta das anteriores para a implementação de máquinas de estado, no âmbito do desenvolvimento de sistemas embebidos em Midas. A abordagem adoptada segue algumas das ideias apresentadas por [Metz et al., 1999], mas faz as devidas adaptações tendo em conta a sintaxe e os mecanismos disponíveis na linguagem Oblog.

6.2 Geração de código Oblog

Nesta secção, indicar-se-ão algumas recomendações que permitem transformar a informação presente num state-chart em código da respectiva classe. O processo de transformação divide-se em duas fases. Em primeiro, fazem reflectir-se, na classe, os comportamentos especificados no respectivo state-chart e, depois, transforma-se a classe em código Oblog.

Relativamente à fig. 6.3(a), admita-se que o atributo volume e a operação ajustarVolume já estavam previamente definidos e que pretende agora fazer repercutir-se nessa classe o respectivo state-chart. A operação ajustarVolume é independente de qualquer estado do objecto e pode, portanto, ser invocada em qualquer instante da vida desse objecto, daí não ser representada no diagrama de estados.

Na primeira fase, adiciona-se um atributo varEstado, cujo valor representa o estado actual do objecto e para cada estado da máquina acrescenta-se um método. Para cada evento, cria-se igualmente um método, que será invocado sempre que esse evento for detectado. Finalmente, para cada acção ou actividade inscrita no state-chart, deve incluir-se um método na classe. A fig. 6.3(b) ilustra a classe em questão depois de introduzidos os atributos e os métodos provenientes do state-chart.

Para melhor organizar listas longas de atributos e operações, pode, antes de cada grupo, usar-se um estereótipo para o classificar [Booch et al., 1999, pág. 52]. Neste contexto, para expressar a proveniência das diversas operações, sugere-se a utilização dos seguintes estereótipos:
≪global≫: operações independentes do state-chart;
≪state≫: operações provenientes dos estados;
≪event≫: operações provenientes dos eventos;
≪action≫: operações provenientes das acções;

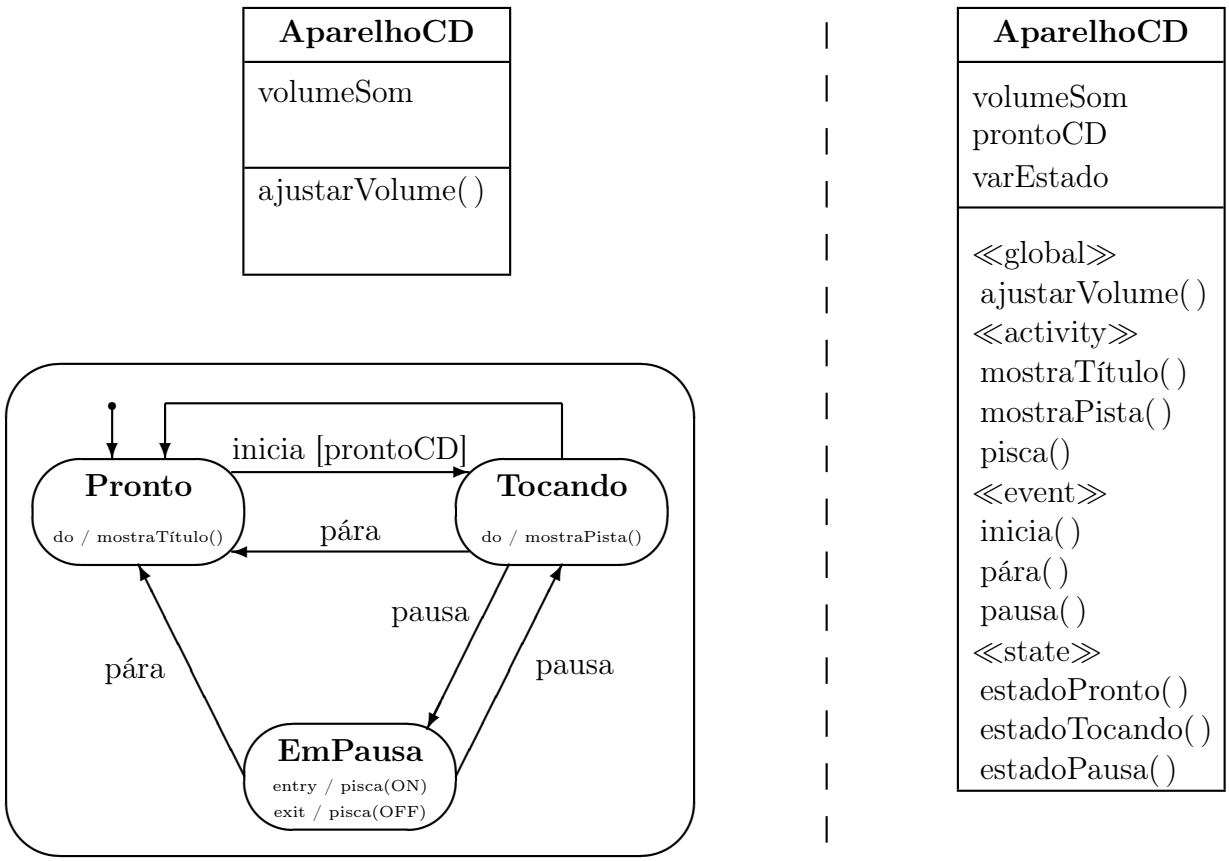


Figura 6.3: O modo como (a) uma classe e o respectivo state-chart se juntam para (b) fazer reflectir na classe o comportamento especificado pelo state-chart [Weber e Metz, 1998].

≪activity≫: operações provenientes das actividades.
Opcionalmente, podem utilizar-se os símbolos '+' (público), '#' (protegido) e '-' (privado), para indicar o tipo de *visibilidade* pretendido para cada propriedade (atributo ou operação) da classe [Booch et al., 1999, pág. 123].

Os state-charts, por vezes, fazem referências a variáveis, algumas das quais podem ser usadas com o intuito de compactar o número de estados. Esta compactação no número de estados, à custa da introdução de variáveis, está fortemente relacionada com a lei de Wirth (*Algorithms + Data Structures = Programs*) [Wirth, 1976], que permite afirmar que o fortalecimento duma das parcelas implica o emagrecimento da outra, ou seja, a introdução de variáveis permite diminuir o número de estados e vice versa. Essas variáveis devem ser adicionadas à classe sob a forma de atributos. Para o exemplo do AparelhoCD, deve ser adicionado, à classe, o atributo prontoCD proveniente do state-chart.

Sempre que um objecto recebe uma mensagem ou evento, o respectivo método é invocado, desde que a consulta do atributo varEstado indique que o objecto pode aceitar essa mensagem no estado actual. Em caso afirmativo, testa-se a condição da transição, caso exista, executam-se as acções (pela ordem correcta, tendo em consideração os contornos atravessados e as acções *entry* e *exit*) e activam-se os eventos indicados na transição. De seguida, deve ser invocado o método do estado destino, onde o valor do atributo varEstado é actualizado.

Seguidamente, apresenta-se o código da classe AparelhoCD, escrito em linguagem Oblog, que permite constatar a forma como os vários métodos se relacionam entre si.

```
 1:  class AparelhoCD
 2:    declarations
 3:      data types
 4:        TypEstado = enum ("Inicial", "Pronto", "Tocando", "EmPausa", "Final") default "Pronto";
 5:      constants
 6:        EstInicial:TypEstado << "Pronto";
 7:        VolMax:Integer << 10;
 8:        VolInicial:Integer << 5;
 9:      event types
10:        INIT;
11:        STOP;
12:        PAUSE;
13:        VOL {valor : Integer;};
14:      exception types
15:        ERRO {msg : String;};
16:      attributes
17:        object
18:          volumeSom : Integer;
19:          prontoCD  : Bool;
20:
21:       // varEstado indica qual o estado actual
22:          varEstado : TypEstado;
23:
24:       // prox indica qual o estado seguinte. Quando prox é diferente de
25:       // varEstado, então há que provocar uma transição de estado
26:          prox      : TypEstado;
27:
28:       // aborta indica que um evento assíncrono foi sentido, enquanto o estado
29:       // actual estava a processar a respectiva actividade (fimActiv = FALSE)
30:          aborta : Bool;
31:
32:       // fimActiv indica que um estado terminou a sua actividade mas que
33:       // ainda não transitou (transição automática ou espera de evento)
34:       // Permite determinar se, aquando dum evento assíncrono, é necessário
35:       // interromper ou não a actividade do estado actual
36:          fimActiv  : Bool;
37:
38:    operations
39:      class
40:        *cria;  // cria instancia
```

```
41:
42:    body
43:      invariants
44:        EstadoValido:
45:          varEstado = "Pronto" OR varEstado = "Tocando" OR
46:          varEstado = "EmPausa"
47:            exception ERRO(msg << "Estado Invalido");
48:      methods
49:        object
50:
51:      // -- Escalonador ----------------------------------------
52:
53:          !escalona is
54:            enabling TRUE;  // assume-se que um relógio regula a execução desta operação
55:          {
56:            // os eventos assíncronos alteram prox
57:            pre (fimActiv = TRUE)
58:              call self.transAutomatica ();  // altera prox
59:            set fimActiv << FALSE;  // eventos duram apenas um ciclo
60:            pre (prox <> varEstado)
61:              call self.mudaEstado();
62:          } end
63:
64:      // -- transAutomatica -----
65:
66:          transAutomatica
67:            method Tocando is
68:              enabling varEstado = "Tocando"
69:            {
70:              set prox << "Pronto";
71:            } end
72:
73:      // -- mudaEstado ----------
74:
75:          mudaEstado
76:            method Pronto is
77:              enabling prox = "Pronto"
78:            {
78:              set aborta << FALSE;
80:              call self.estadoPronto();
81:            } end
82:
83:          mudaEstado
84:            method Tocando is
85:              enabling prox = "Tocando"
86:            {
87:              set aborta << FALSE;
88:              call self.estadoTocando();
89:            } end
90:
91:          mudaEstado
92:            method EmPausa is
93:              enabling prox = "EmPausa"
94:            {
95:              set aborta << FALSE;
96:              call self.estadoEmPausa();
97:            } end
98:
99:      // -- Métodos para Tratamento de Eventos ----------------
96:
97:      // -- inicia --------------
98:
99:          ^inicia trigger (ev:INIT)
 00:            method Pronto is
101:              enabling varEstado = "Pronto" and prontoCD
102:            {
103:              pre (fimActiv = FALSE)
104:                set aborta << TRUE;
105:              set prox << "Tocando";
106:            } end
107:
108:      // -- para ----------------
109:
```

```
110:          ^para trigger (ev:STOP)
111:            method EmPausa is
112:              enabling varEstado = "EmPausa"
113:            {
114:              pre (fimActiv = FALSE)
115:                set aborta << TRUE;
116:              call self.pisca(OFF);  // executa acção exit de EmPausa
117:              set prox << "Pronto";
118:            } end
119:
120:          ^para trigger (ev:STOP)
121:            method Tocando is
122:              enabling varEstado = "Tocando"
123:            {
124:              pre (fimActiv = FALSE)
125:                set aborta << TRUE;
126:              set prox << "Pronto";
127:            } end
128:
129:      // -- pausa ------------------------------------------
130:
131:          ^pausa trigger (ev:PAUSE)
132:            method EmPausa is
133:              enabling varEstado = "EmPausa"
134:            {
135:              pre (fimActiv = FALSE)
136:                set aborta << TRUE;
137:              call self.pisca(OFF);  // executa acção exit de EmPausa
138:              set prox << "Tocando";
139:            } end
140:
141:          ^pausa trigger (ev:PAUSE)
142:            method Tocando is
143:              enabling varEstado = "Tocando"
144:            {
145:              pre (fimActiv = FALSE)
146:                set aborta << TRUE;
147:              set prox << "EmPausa";
148:            } end
149:
150:       // -- ajustarVolume -------
151:
152:          ^ajustarVolume trigger (ev:VOL) is
153:           {
154:            pre (ev.valor >= 0 and ev.valor <= VolMax)
155:              set volumeSom << ev.valor;
156:          } end
157:
158:      // -- Métodos para Estados -----------------------------
159:
160:          estadoPronto is
161:          {
162:            set varEstado << "Pronto";
163:            { assert (aborta = FALSE);
164:              call self.mostraTitulo(); }  // executa actividade de Pronto
165:            set fimActiv << TRUE;
166:          } end
167:
168:          estadoTocando is
169:          {
170:            set varEstado << "Tocando";
171:            { assert (aborta = FALSE);
172:              call self.mostraPista(); }  // executa actividade de Tocando
173:            set fimActiv << TRUE;          // transição automática
174:          } end
175:
176:          estadoEmPausa is
177:          {
178:            set varEstado << "EmPausa";
179:            { assert (aborta = FALSE);
180:              call self.pisca(ON); }  // executa acção entry de EmPausa
181:            set fimActiv << TRUE;
182:          } end
```

```
183:
184:      // -- Métodos para Actividades -------------------------
185:
186:          mostraTitulo is
187:          {
188:            // terminar
189:          } end
190:
191:          mostraPista is
192:          {
193:            // terminar
194:          } end
195:
196:          pisca is
197:          {
198:            // terminar
199:          } end
200:
201:      // -- Tratamento de Excepções --------------------------
202:
203:      // -- Erro ----------------
204:
205:          ^ErroExc trigger (exc:ERRO)
206:          {
207:            set varEstado << EstInicial;
208:          } end
209:
210:        class
211:
212:      // -- cria ---------------------------------------------
213:
214:          *cria is
215:          {
216:            set volumeSom << VolInicial;
217:            set prontoCD  << FALSE;
218:            set varEstado << "Inicial";
219:            set prox      << EstInicial;
220:            set aborta    << FALSE;
221:            set fimActiv  << FALSE;
222:          } end
223:
224:  end class
```

Cada objecto descrito, na linguagem OBLOG, pode pressupor vários fios de execução, ou visto duma outra perspectiva não consiste necessariamente num objecto sequencial.

O tipo enumerado TypEstado permite criar um atributo varEstado que indica o estado actual em que o objecto se encontra. Os valores possíveis para o atributo varEstado incluem, além dos estados válidos em que esse objecto pode estar, os valores "Inicial" e "Final".

A operação escalona é do tipo iniciativa própria (*self-initiative*), pelo que está sempre em execução. É da sua responsabilidade verificar, quando ocorre um pulso no sinal de relógio, se há que proceder a uma transição de estado e, em caso afirmativo, invocar o respectivo método. As transições de estado podem dar-se por duas causas.

A primeira causa verifica-se quando surge um evento ao qual o objecto é sensível no estado actual. Nesse caso, as operações que tratam eventos alteram o valor do atributo prox. A operação escalona, sempre que detecta uma diferença nos valores dos atributos varEstado (que indica o estado actual) e prox (que indica o próximo estado), invoca o método mudaEstado provocando uma transição de estado. A segunda causa resulta quando a actividade do estado actual terminou, indicada pelo atributo fimActiv, quando retém o valor TRUE. Desta forma, a operação escalona verifica ciclicamente (i.e. em cada pulso do sinal de relógio) se alguma transição automática está habilitada a disparar (pode não o estar pelo facto da respectiva condição ser avaliada como falsa).

Foram usadas condições de habilitação (*enabling conditions*) para as operações, por forma a restringir a validade dos métodos aos estados em que a sua invocação é permitida. De notar que, em Oblog, uma operação pode ser dividida em vários métodos[1], desde que as respectivas condições de habilitação sejam mutuamente exclusivas. Este mecanismo permite muito facilmente introduzir novas realidades para uma dada operação (por exemplo, a sua invocação num outro estado), sem implicar a alteração da realidade anterior, mostrando-se útil, por exemplo, para suportar novos comportamentos nas subclasses, quando se usam relações hierárquicas entre classes.

Repare-se que os métodos que tratam eventos executam assincronamente e assume-se aqui que o fazem de forma concorrente com os outros métodos em eventual execução. Os eventos podem surgir em qualquer instante e implicam, em caso de transição, que o estado actual deva ser alterado e que a actividade que possa estar em execução deva ser imediatamente interrompida. Esta situação é modelada através duma excepção que é activada sempre que o atributo aborta não é avaliado com o valor FALSE (i.e. sempre que é avaliado com o valor TRUE).

6.3 Casos típicos de modelação

O exemplo do AparelhoCD considerado foi propositadamente simples, uma vez que se pretendia mostrar a filosofia genérica de escrita de código com base nos state-charts. Nesta secção, apresentam-se alguns casos de modelação mais comuns, mostrando o respectivo código Oblog que deve ser escrito. Note-se que não são tratados todos os mecanismos de modelação previstos no meta-modelo state-chart, embora fosse possível fazê-lo por uma questão de exaustão. Alguns dos mecanismos não considerados são os seguintes: transições internas a estados (*internal transitions*), eventos adiados (*deferred events*) e conector história profundo (*deep history state*). No final desta secção, apresenta-se um exemplo mais elaborado, em que alguns dos casos típicos de modelação são combinados, a fim de reproduzir o comportamento descrito pelo state-chart.

6.3.1 Transição Inicial

Cada máquina de estados deve possuir uma transição inicial, que indica qual o primeiro estado a estar activo (*default state*), assim que o respectivo objecto é criado. Ao nível mais alto da hierarquia, a transição inicial não tem associado qualquer evento ou condição (pode unicamente associar-se o estereótipo ≪create≫, para explicitar que se trata da transição a disparar aquando da criação do objecto).

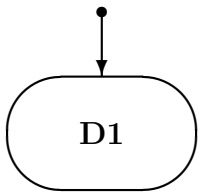


Figura 6.4: Transição Inicial.

[1]Os termos "operação" e "método" têm, em Oblog, significados ligeiramente distintos dos que até agora foram usados. Em Oblog, uma *operação* corresponde à implementação duma funcionalidade do objecto, enquanto que um *método* representa uma parte da implementação duma operação.

A transição inicial, em OBLOG, é codificada da forma que segue:

```
 1:  declarations
 2:    constants
 3:      EstInicial:TypEstado << "D1"
 4:
 5:  // -- mudaEstado ----------
 6:
 7:  mudaEstado
 8:    method D1 is
 9:      enabling prox = "D1"
10:    {
11:      set aborta << FALSE;
12:      call self.estadoD1();
13:    } end
14:
15:  // -- cria ---------------------------------------------
16:
17:  *cria is
18:  {
19:    set varEstado << "Inicial";
20:    set prox << EstInicial;
21:    // terminar
22:  } end
```

Repare-se que neste caso, aquando da criação do objecto, os atributos varEstado e prox têm valores diferentes, pelo que o método escalona irá invocar o método mudaEstado, colocando assim a máquina no estado inicial D1.

6.3.2 Acções e Actividades

A fig. 6.5 mostra dois estados ligados por uma transição sensível ao evento evB.

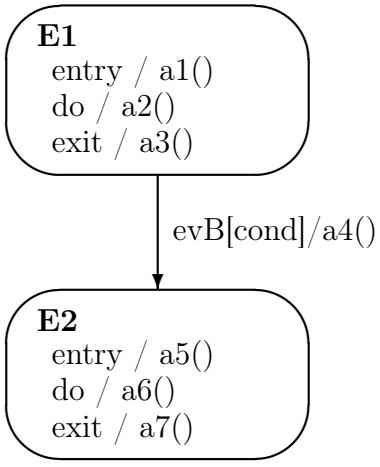


Figura 6.5: State-chart em que se usam acções e actividades.

O código para esta situação de modelação é apresentado de seguida. Repare-se que as acções *entry* e as actividades são indicadas, apenas uma vez, nos métodos dos respectivos estados, enquanto que as acções da transição e as acções *exit* são referidas nos métodos do evento responsável por disparar a referida transição. Se existissem outras transições, originárias de E1, sensíveis a outros eventos, nos métodos que tratam estes, seria novamente necessário incluir a chamada à acção *exit* (a3).

```
 1:  // -- Métodos para Estados -------------------------
 2:
 3:  estadoE1 is
 4:  {
 5:    set varEstado << "E1";
 6:    call self.a1();  // executa acção entry de E1
 7:    { assert (aborta = FALSE);
 8:      call self.a2(); }  // executa actividade de E1
 9:    set fimActiv << TRUE;
10:  } end
11:
12:  estadoE2 is
13:  {
14:    set varEstado << "E2";
15:    call self.a5();  // executa acção entry de E2
16:    { assert (aborta = FALSE);
17:      call self.a6(); }  // executa actividade de E2
18:    set fimActiv << TRUE;
19:  } end
20:
21:  // -- Métodos para Eventos -------------------------
22:
23:  ^trataEvB trigger (evB:Event) is
24:    method E1 is
25:      enabling varEstado = "E1" and cond
26:    {
27:      pre (fimActiv = FALSE)
28:        set aborta << TRUE;
29:      call self.a3();  // executa acção exit de E1
30:      call self.a4();  // executa acção da transição
31:      set prox << "E2";
32:    } end
```

O comando `assert (aborta=FALSE)` (linha 7) testa se o atributo aborta tem o valor FALSE. Se esse teste for violado, enquanto a actividade do estado E1 (a2) estiver a executar, esta tem de ser imediatamente interrompida, devido à ocorrência do evento assíncrono evB, relevante no estado E1. Repare-se que o atributo aborta só é colocado com o valor a TRUE (linha 28), que fará violar a condição de *assert*, se a actividade não tiver terminado (este facto é indicado pelo atributo fimActiv com o valor a FALSE na linha 27).

6.3.3 Transições atravessando vários contornos

Por vezes, há transições que atravessam vários contornos de estados até chegarem ao seu destino. Nessas situações, há que ter em atenção todos os contornos atravessados pela transição e indicar, pela ordem correcta, as diversas acções e actividades.

A fig. 6.6 apresenta uma situação em que uma transição atravessa vários contornos. Note-se que os super-estados F1 e F2 contêm acções *entry* e *exit*, não sendo todavia permitido, pela sintaxe do formalismo state-chart, a indicação de actividades.

O código para esta situação de modelação é apresentado de seguida.

```
 1:  // -- Métodos para Estados -------------------------
 2:
 3:  estadoF11 is
 4:  {
 5:    set varEstado << "F11";
 6:    call self.a1();  // executa acção entry de F11
 7:    { assert (aborta = FALSE);
 8:      call self.a2(); }  // executa actividade de F11
 9:    set fimActiv << TRUE;
10:  } end
11:
```

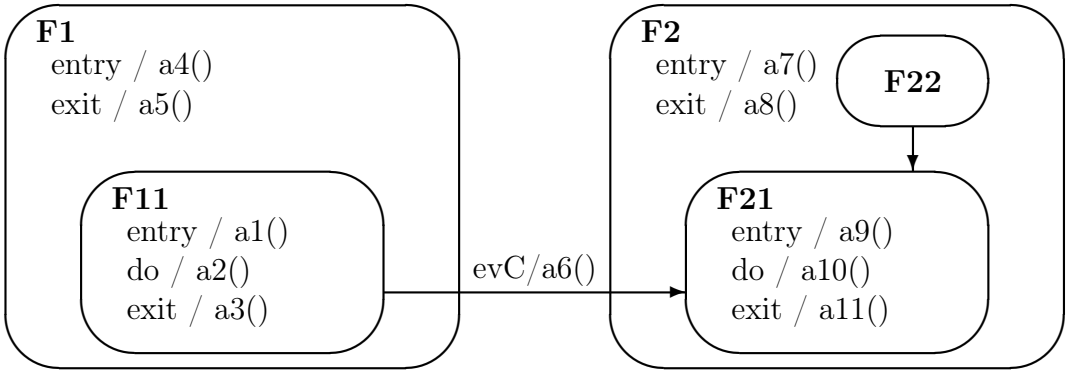


Figura 6.6: State-chart em que uma transição atravessa vários contornos.

```
12:  estadoF21 is
13:  {
14:    set varEstado << "F21";
15:    call self.a9();   // executa acção entry de F21
16:    { assert (aborta = FALSE);
17:      call self.a10(); }  // executa actividade de F21
18:    set fimActiv << TRUE;
19:  } end
20:
21:  // -- Métodos para Eventos --------------------------
22:
23:  ^trataEvC trigger (evC:Event)
24:    method F11 is
25:      enabling varEstado = "F11"
26:    {
27:      pre (fimActiv = FALSE)
28:        set aborta << TRUE;
29:      call self.a3();  // executa acção exit de F11
30:      call self.a5();  // executa acção exit de F1
31:      call self.a6();  // executa acção da transição
32:      call self.a7();  // executa acção entry de F2
33:      set prox << "F21";
34:    } end
```

Repare-se no facto da acção *entry* de F2 (a7) ser executada no contexto do método que trata o evento evC (linha 32) e não no método do estado F21. Esta solução foi adoptada, pois o estado F21 pode ser alcançado por transições internas ao contexto de F2 (por exemplo uma transição de F22 para F21), o que não implica a execução da acção *entry* de F2.

6.3.4 Transições automáticas

Outra situação a considerar refere-se às transições automáticas (*completion transitions*), aquelas que não têm associado qualquer evento. O disparo duma transição automática faz-se assim que terminar a actividade associada ao estado origem e tendo em consideração a eventual condição da transição. A fig. 6.7 ilustra duas transições automáticas. Na fig. 6.7(a) não há condição associada à transição, o que já sucede na fig. 6.7(b).

Em state-charts, a finalização das actividades dum estado implica, caso exista uma transição automática, o disparo desta, pelo que este mecanismo de modelação deve integrar-se, em OBLOG, nos métodos de estado. Assim, o método do estado origem (duma transição automática) deve executar a actividade e a acção *exit* que lhe estão associadas e, posteriormente, colocar o atributo fimActiv com o valor TRUE. O método escalona é sensível a este valor e determina que se

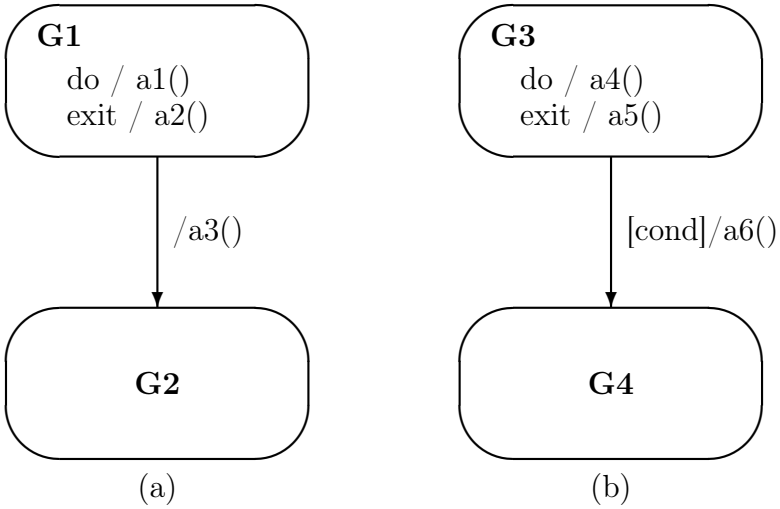


Figura 6.7: State-charts em que se usam transições automáticas.

invoque um método da operação transAutomatica, responsável por executar a acção da transição e por indicar qual o estado destino da transição automática. O código para esta situação de modelação é apresentado de seguida.

```
 1:  // -- transAutomatica -----
 2:
 3:  transAutomatica
 4:    method G1 is
 5:      enabling varEstado = "G1"
 6:    {
 7:      call self.a2();  // executa acção exit de G1
 8:      call self.a3();  // executa acção da transição
 9:      set prox << "G2";
10:    } end
11:
12:  transAutomatica
13:    method G3 is
14:      enabling varEstado = "G3" and cond
15:    {
16:      call self.a5();  // executa acção exit de G3
17:      call self.a6();  // executa acção da transição
18:      set prox << "G4";
19:    } end
20:
21:  // -- Métodos para Estados --------------------------
22:
23:  estadoG1 is
24:  {
25:    set varEstado << "G1";
26:    { assert (aborta = FALSE);
27:      call self.a1(); }  // executa actividade de G1
28:    set fimActiv << TRUE;
29:  } end
30:
31:  estadoG2 is
32:  {
33:    set varEstado << "G2";
34:    // terminar
35:  } end
36:
37:  estadoG3 is
38:  {
39:    set varEstado << "G3";
40:    { assert (aborta = FALSE);
41:      call self.a4(); }  // executa actividade de G3
42:    set fimActiv << TRUE;
```

```
43:  } end
44:
45:  estadoG4 is
46:  {
47:    set varEstado << "G4";
48:    // terminar
49:  } end
```

6.3.5 Transições condicionais

No meta-modelo state-chart, é possível uma transição ser dividida em dois ou mais ramos, associando-se, a cada um deles, uma condição. Em MIDAS, as várias condições têm de ser obrigatoriamente disjuntas entre si, de modo a garantir que a máquina de estados tem um comportamento determinístico. Se as condições não forem totais, então, caso nenhuma se verifique, não há qualquer mudança de estado. Na fig. 6.8 apresentam-se dois exemplos de state-charts em que as transições são divididas em dois ramos.

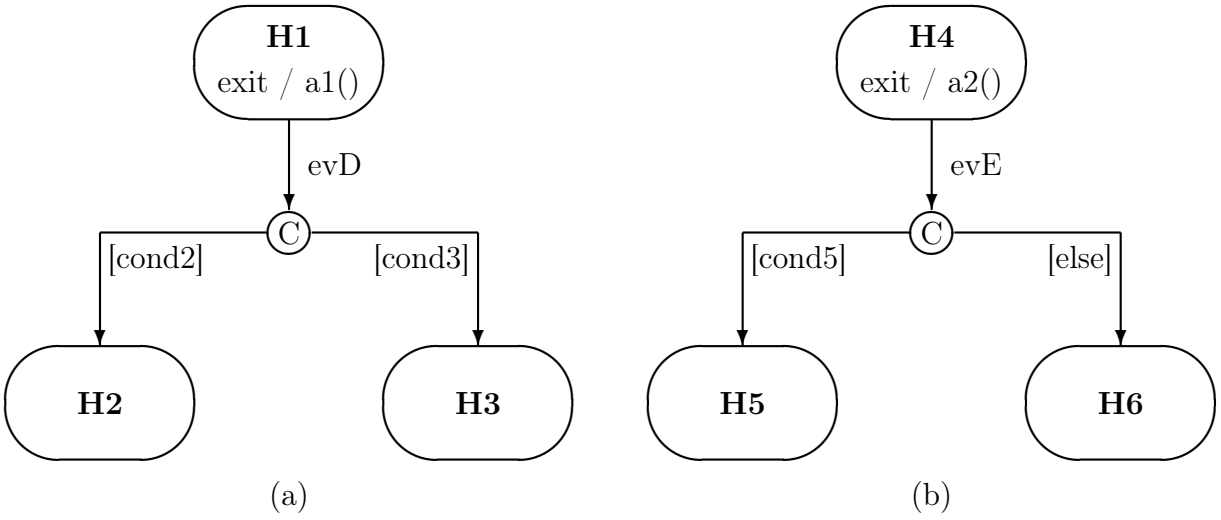


Figura 6.8: State-chart em que se usam transições com ramos.

Relativamente à fig. 6.8(a), o código respeitante ao evento evD é o seguinte:

```
 1:  // -- Métodos para Eventos --------------------------
 2:
 3:  ^trataEvD trigger (evD:Event)
 4:    method H1a is
 5:      enabling varEstado = "H1" and cond2
 6:    {
 7:      pre (fimActiv = FALSE)
 8:        set aborta << TRUE;
 9:      call self.a1();  // executa acção exit de H1
10:      set prox << "H2";
11:    } end
12:
13:  ^trataEvD trigger (evD:Event)
14:    method H1b is
15:      enabling varEstado = "H1" and cond3
16:    {
17:      pre (fimActiv = FALSE)
18:        set aborta << TRUE;
19:      call self.a1();  // executa acção exit de H1
20:      set prox << "H3";
21:    } end
```

Em UML, é possível indicar uma das condições como sendo [else], significando tal que se nenhuma das outras condições for verdadeira, então aquela condição assume esse valor. Este facto permite garantir que, em qualquer circunstância, a transição será disparada, motivo pelo qual há a certeza que a acção *exit* do estado origem será sempre executada. Relativamente à fig. 6.8(b), o código para o evento evE é o seguinte:

```
// -- Métodos para Eventos --------------------------

^trataEvE trigger (evE:Event)
  method H4a is
    enabling varEstado = "H4" and cond5
  {
    pre (fimActiv = FALSE)
      set aborta << TRUE;
    call self.a2();  // executa acção exit de H4
    set prox << "H5";
  } end

^trataEvE trigger (evE:Event)
  method H4b is
    enabling varEstado = "H4" and not cond5
  {
    pre (fimActiv = FALSE)
      set aborta << TRUE;
    call self.a2();  // executa acção exit de H4
    set prox << "H6";
  } end
```

Se além do ramo [else], existirem n ramos ($n \geq 1$), cada um dos quais com a condição $C_i : i \in \{1 \ldots n\}$, então a condição a colocar na condição de habilitação do método para o evento, relativo ao ramo [else] é: $\neg \bigwedge_{i=1}^{n} C_i$.

6.3.6 Transições com eventos temporais

Outro mecanismo de modelação muito usado, sobretudo em sistemas de tempo-real, são os eventos temporais. Estes eventos representam-se em UML através da expressão "tm(T)", em que T representa um dado valor temporal. Um evento temporal tm(T) dispara num estado, se, após o sistema entrar nesse estado, não ocorrer nenhum dos outros eventos a que o sistema é sensível no estado considerado, durante o tempo T.

Na fig. 6.9, mostra-se um exemplo dum state-chart em que uma transição é sensível a um evento temporal. Se no estado I1, a actividade a1 não terminar 5s após se ter iniciado, o estado seguinte será I2. Caso contrário, o evento "fim de actividade" (*completion event*) ocorre antes do evento temporal, o que habilita o disparo da transição automática que termina em I3.

O código Oblog relativo a este exemplo de modelação é apresentado de seguida.

```
declarations
  attributes
    object
      temp : Bool;

// -- transAutomatica -----

transAutomatica
  method I1 is
    enabling varEstado = "I1"
  {
    set prox << "I3";
  } end
```

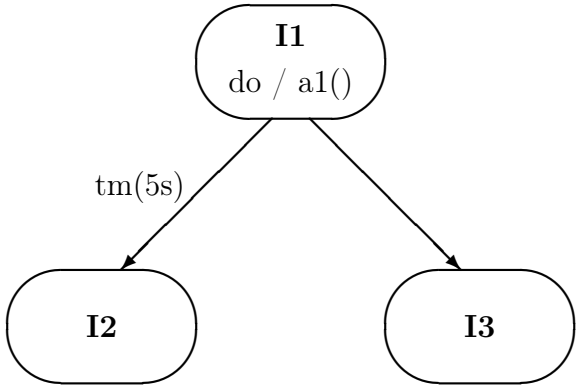


Figura 6.9: State-chart em que se usam transições com eventos temporais.

```
14:
15:  // -- Métodos para Estados -------------------------
16:
17:  estadoI1 is
18:  {
19:    set varEstado << "I1";
20:    set temp << TRUE;
21:    call relogio.start(5);
22:    { assert (aborta = FALSE);
23:      call self.a1(); }  // executa actividade de I1
24:    set fimActiv << TRUE;
25:    set temp << FALSE;
26:  } end
27:
28:  // -- Métodos para Eventos -------------------------
29:
30:  ^trataTm5 trigger (tm5:Event)
31:    method  is
32:      enabling varEstado = "I1" and temp = TRUE
33:    {
34:      pre (fimActiv = FALSE)
35:        set aborta << TRUE;
36:      set prox << "I2";
37:    } end
```

Para tratar os eventos temporais, assume-se a existência dum objecto relógio que recebe inicialmente o tempo a contar (linha 21). Este objecto, quando esse tempo se esgotar, dispara o evento tm5 que é tratado pelo método trataTm5. Este método só será executado se o atributo temp estiver a TRUE, o que se verifica caso a actividade a1 ainda não tiver terminado (linhas 20, 23 e 25).

6.3.7 Transições de grupo

No meta-modelo state-chart, é possível uma transição iniciar-se no contorno dum super-estado, o que indica que essa transição se aplica a todos os sub-estados desse super-estado. Na fig. 6.10, apresenta-se um exemplo dum state-chart em que uma transição provém do contorno dum super-estado (transição de grupo).

O código para o método relativo ao evento evG é o que se apresenta de seguida.

```
1:  // -- Métodos para Eventos -------------------------
2:
3:  ^trataEvG trigger (evG:Event)
```

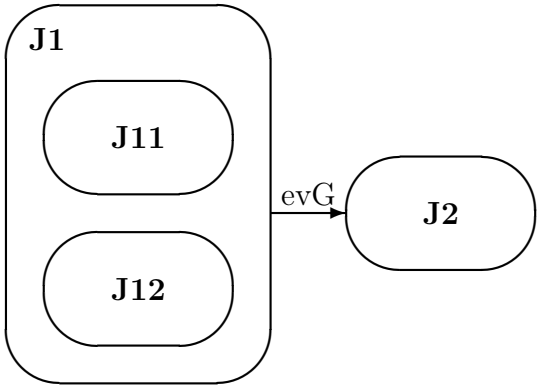


Figura 6.10: State-chart em que se usa uma transição de grupo.

```
 4:    method J11 is
 5:      enabling varEstado = "J11"
 6:    {
 7:      pre (fimActiv = FALSE)
 8:        set aborta << TRUE;
 9:      set prox << "J2";
10:    } end
11:
12:  ^trataEvG trigger (evG:Event)
13:    method J12 is
14:      enabling varEstado = "J12"
15:    {
16:      pre (fimActiv = FALSE)
17:        set aborta << TRUE;
18:      set prox << "J2";
19:    } end
```

Repare-se que neste caso, a solução assenta em criar tantos métodos para o tratamento do evento, quantos os estados internos a J1 (no caso 2). Caso se excluam as condições de habilitação (linhas 5 e 14), os métodos são iguais. Assim sendo, os varios métodos alternativos poderiam ser agrupados num só, se as acções *exit* forem as mesmas (inclui-se também, neste caso, a situação de não haver a indicação de qualquer acção *exit*). Contudo, a divisão dos métodos proposta (um por cada estado) mostra-se mais adequada pois facilita a alteração do código, quando se fazem modificações nos state-charts.

6.3.8 Transição para super-estado

O exemplo da fig. 6.11 pretende ilustrar uma transição que termina num contorno dum super-estado, o que obriga a determinar qual o estado inicial, com base na informação interna ao super-estado.

A codificação deste caso de modelação foi feita tendo em conta que é conveniente o código ser o mais flexível possível (i.e. ser facilmente alterado, caso o state-chart também o seja). Assim, teria sido possível escrever, na linha 11, o nome do estado inicial de K1 (no caso, K11). No entanto, caso haja alguma alteração relativamente ao estado inicial de K1, a solução adoptada mostra-se mais adequada, pois tem unicamente que proceder-se a uma alteração num local bem determinado (linha 3).

```
1:  declarations
2:    constants
```

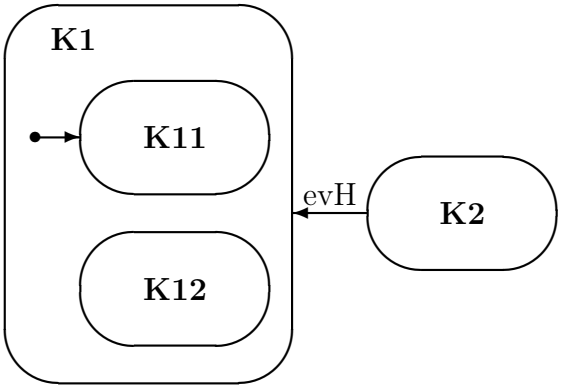


Figura 6.11: State-chart onde uma transição termina no contorno dum super-estado.

```
 3:      EstInicialK1:TypEstado << "K11"
 4:
 5:  // -- Métodos para Eventos --------------------------
 6:
 7:  ^trataEvH trigger (evH:Event)
 8:    method K2 is
 9:      enabling varEstado = "K2"
10:    {
11:      pre (fimActiv = FALSE)
22:        set aborta << TRUE;
13:      set prox << EstInicialK1;
14:    } end
```

6.3.9 Conectores história

O exemplo da fig. 6.12 pretende ilustrar uma transição que termina num conector história, o que obriga a determinar qual o estado destino.

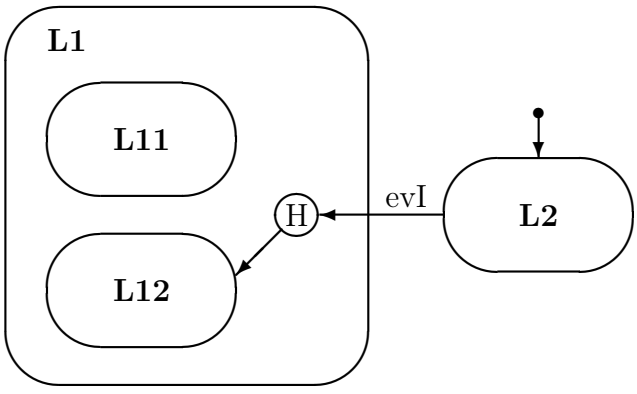


Figura 6.12: State-chart em que se usa um conector história.

Para codificar esta situação, é necessário um atributo (varHistoriaL1) que guarda o último estado activo no contexto de L1. Este atributo é actualizado sempre que algum dos sub-estados de L1 (no caso L11 e L12) for o estado actual (linhas 21 e 29). Assim, qualquer transição que termine num conector história deve primeiramente consultar esse atributo, para determinar qual o estado para onde deve transitar (linha 13).

```
 1:  declarations
 2:    attributes
 3:      varHistoriaL1:TypEstado << "L12"
 4:
 5:  // -- Métodos para Eventos -------------------------
 6:
 7:  ^trataEvI trigger (evI:Event)
 8:    method L2 is
 9:      enabling varEstado = "L2"
10:    {
11:      pre (fimActiv = FALSE)
12:        set aborta << TRUE;
13:      set prox << varHistoriaL1;
14:    } end
15:
16:  // -- Métodos para Estados -------------------------
17:
18:  estadoL11 is
19:  {
20:    set varEstado << "L11";
21:    set varHistorialL1 << "L11";
22:    // terminar
23:    set fimActiv << TRUE;
24:  } end
25:
26:  estadoL12 is
27:  {
28:    set varEstado << "L12";
29:    set varHistorialL1 << "L12";
30:    // terminar
31:    set fimActiv << TRUE;
32:  } end
33:
34:  estadoL2 is
35:  {
36:    set varEstado << "L2";
37:    // terminar
38:    set fimActiv << TRUE;
39:  } end
```

Em UML, existe uma variante deste conector (conector história profundo) em que é memorizado o último estado activo de todos os contextos interiores àquele em que o conector é indicado (e não estritamente nesse contexto). A solução para este mecanismo é semelhante à apresentada anteriormente e assenta igualmente no recurso a um atributo para memorizar o estado activo, valor esse que é actualizado em todos os métodos dos estados em que o conector é válido (no caso, todos os estados e sub-estados do contexto em que o conector é indicado).

6.3.10 State-charts hierárquicos

O exemplo da fig. 6.13 pretende ilustrar uma situação em que se usam sub-máquinas, o que pressupõe que o state-chart dum dado objecto é especificado de forma hierárquica. No processo de codificação, pretende-se que o código mantenha a mesma estrutura hierárquica que o state-chart, apesar de ser válido "desfazer" a hierarquia e codificar a máquina de estados plana equivalente. Contudo, dessa forma estariam a criar-se as condições para dificultar o relacionamento, que se pretende simples e directo, entre os diagramas e o código. Por exemplo, a alteração dum diagrama state-chart poderia implicar a modificação em diversos locais distintos do código, o que obviamente não se pretende.

Uma sub-máquina é uma conveniência sintáctica que facilita a reutilização e a modularidade. Trata-se duma simplificação notacional que implica uma expansão estrutural do estado que inclui a chamada à sub-máquina. Note-se que a mesma sub-máquina pode ser referida mais

do que uma vez no contexto dum único state-chart, representando cada estado em que a sub-máquina é referida algo semelhante à chamada duma subrotina.

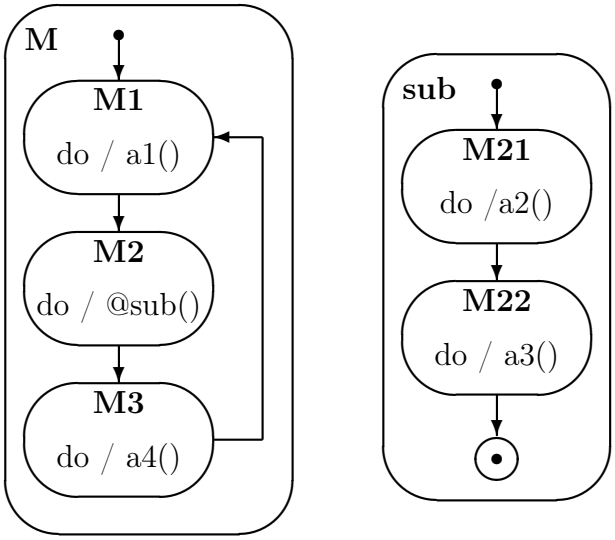


Figura 6.13: State-chart hierárquico.

Uma solução, para codificar esta situação, recorre a uma estrutura de dados que armazena o estado para o qual se deve transitar, assim que terminar um dado state-chart. Desta forma, a exemplo do que sucede na programação com linguagens de alto nível (tipo C, PASCAL), podem ter-se vários níveis de hierarquia. O acesso à estrutura de dados deve obedecer ao princípio LIFO (*last-in first-out*) e pode ser implementado, por exemplo, por uma pilha (*stack*). Para introduzir e retirar um estado na estrutura de dados, devem ser introduzidos os métodos guarda e recupera, respectivamente. Antes de transitar para a sub-máquina, o método guarda é invocado, tendo como parâmetro o estado seguinte ao do estado em que a sub-máquina é referida. Desta forma, é possível quando terminar essa sub-máquina, invocar o método recupera que devolve como resultado o estado seguinte no nível imediatamente superior ao da sub-máquina actual.

```
 1:  declarations
 2:    constants
 3:      EstInicialM:TypEstado << "M1"
 4:      EstInicialsub:TypEstado << "M21"
 5:
 6:  // -- transAutomatica -----
 7:
 8:  transAutomatica
 9:    method M1 is
10:      enabling varEstado = "M1"
11:    {
12:      set prox << "M2";
13:    } end
14:
15:  transAutomatica
16:    method M21 is
17:      enabling varEstado = "M21"
18:    {
19:      set prox << "M22"
20:      set fimActiv << FALSE;
21:    } end
22:
23:  transAutomatica
24:    method M22 is
25:      enabling varEstado = "M22"
```

```
26:    {
27:      set prox << recupera();  // recupera, em M, estado seguinte a M2
28:      set fimActiv << FALSE;
29:    } end
30:
31:  transAutomatica
32:    method M3 is
33:      enabling varEstado = "M3"
34:    {
35:      set prox << "M1";
36:      set fimActiv << FALSE;
37:    } end
38:
39:  // -- Métodos para Estados --------------------------
40:
41:  estadoM1 is
42:  {
43:    set varEstado << "M1";
44:    { assert (aborta = FALSE);
45:      call self.a1(); }  // executa actividade de M1
46:    set fimActiv << TRUE;
47:  } end
48:
49:  estadoM2 is
50:  {
51:    set varEstado << "M2";
52:    set prox << EstInicialsub;
53:    guarda("M3");  // guarda estado seguinte em M
54:    set fimActiv << TRUE;
55:  } end
56:
57:  estadoM21 is
58:  {
59:    set varEstado << "M21";
60:    { assert (aborta = FALSE);
61:      call self.a2(); }  // executa actividade de M21
62:    set fimActiv << TRUE;
63:  } end
64:
65:  estadoM22 is
66:  {
67:    set varEstado << "M22";
68:    { assert (aborta = FALSE);
69:      call self.a3(); }  // executa actividade de M22
70:    set fimActiv << TRUE;
71:  } end
72:
73:  estadoM3 is
74:  {
75:    set varEstado << "M3";
76:    { assert (aborta = FALSE);
77:      call self.a4(); }  // executa actividade de M3
78:    set fimActiv << TRUE;
79:  } end
```

O exemplo apresentado envolve apenas transições automáticas, pelo que a máquina de estados não precisa de activar o pseudo-estado final da sub-máquina relativa ao estado M2. Apesar do meta-modelo state-chart permitir outras hipóteses, nomeadamente a aposição de etiquetas às transições que terminam num pseudo-estado final da sub-máquina e às transições que saem do estado em que uma sub-máquina é invocada, essas possibilidades não serão abordadas, uma vez que a filosofia genérica de geração de código Oblog é, nesses casos, semelhante à já apresentada.

Estrutura vs. código

Existem, pelo menos, duas alternativas para codificar em OBLOG uma sub-máquina. A primeira alternativa (*estrutural*) consiste em considerar que a sub-máquina é uma extensão estrutural do state-chart onde é referida e, neste caso, a sub-máquina pode substituir o estado de nível superior em que aquela é invocada. A segunda alternativa (*comportamental*) contempla a sub-máquina como a especificação da sequência de instruções do estado que encapsula essa sub-máquina. Retomando a fig. 6.13, a codificação apresentada anteriormente para esse state-chart seguiu a alternativa estrutural, pelo que, de seguida, se mostra o código segundo a alternativa comportamental para a sub-máquina relativa ao estado M2.

```
 1:  // -- transAutomatica -----
 2:
 3:  transAutomatica
 4:    method M1 is
 5:      enabling varEstado = "M1"
 6:    {
 7:      set prox << "M2";
 8:      set fimActiv << FALSE;
 9:    } end
10:
11:  transAutomatica
12:    method M2 is
13:      enabling varEstado = "M2"
14:    {
15:      set prox << "M3"
16:      set fimActiv << FALSE;
17:    } end
18:
19:  transAutomatica
20:    method M3 is
21:      enabling varEstado = "M3"
22:    {
23:      set prox << "M1";
24:      set fimActiv << FALSE;
25:    } end
26:
27:  // -- Métodos para Estados --------------------------
28:
29:  estadoM1 is
30:  {
31:    set varEstado << "M1";
32:    { assert (aborta = FALSE);
33:      call self.a1(); }  // executa actividade de M1
34:    set fimActiv << TRUE;
35:  } end
36:
37:  estadoM2 is
38:  {
39:    set varEstado << "M2";
40:    { assert (aborta = FALSE);
41:      call self.a2();    // executa actividade de M21
42:      call self.a3(); }  // executa actividade de M22
43:    set fimActiv << TRUE;
44:  } end
45:
46:  estadoM3 is
47:  {
48:    set varEstado << "M3";
49:    { assert (aborta = FALSE);
50:      call self.a4(); }  // executa actividade de M3
51:    set fimActiv << TRUE;
52:  } end
```

Neste caso, deixam de ser necessários os métodos guarda e recupera e, no interior do método correspondente ao estado M2, codifica-se o algoritmo descrito pela máquina de estados (linhas 41

e 42). A perspectiva comportamental pode ser levada ao extremo, dando origem ao seguinte código, novamente para o state-chart apresentado na fig. 6.13.

```
 1:  // -- Métodos para Estados -------------------------
 2:
 3:  estadoM is
 4:  {
 5:    set varEstado << "M";
 6:    { assert (aborta = FALSE);
 7:      call self.a1();    // executa actividade de M1
 8:      call self.a2();    // executa actividade de M21
 9:      call self.a3();    // executa actividade de M22
10:      call self.a4(); }  // executa actividade de M3
11:    set fimActiv << TRUE;
12:  } end
```

Como atrás se viu, é possível combinar as duas alternativas de codificação para o mesmo objecto/sistema especificado por um state-chart composto por várias sub-máquinas relacionadas hierarquicamente, considerando que se segue a primeira alternativa até um dado nível hierárquico e que daí até às folhas da hierarquia se segue a segunda alternativa. Contudo, não é simples estabelecer uma regra firme que indique até que nível hierárquico se deve aplicar a primeira alternativa. Na prática, será da responsabilidade da equipa de projecto tomar essa decisão com base nas características do state-chart (i.e. das várias sub-máquinas que o constituem).

Em Midas, sugere-se a utilização do prefixo '@' no nome da actividade associada a um estado, por forma a indicar que essa actividade é especificada com um state-chart (no caso uma sub-máquina) que corresponde a uma extensão estrutural da máquina. A ausência do prefixo será interpretada com o significado de a actividade ser especificada directamente por código, terminando assim esse ramo da estrutura hierárquica dos state-charts.

6.3.11 Exemplo

O exemplo da fig. 6.14 mostra um state-chart, a partir do qual se pretende obter directamente o respectivo código Oblog.

O código Oblog para este state-chart, obtém-se por combinação de alguns dos mecanismos de modelação apresentados anteriormente. O código completo é apresentado de seguida:

```
 1:  class StateChart
 2:    declarations
 3:      data types
 4:        TypEstado = enum ("E11", "E12", "E2", "E31", "E32") default "E11";
 5:      constants
 6:        EstInicialE1:TypEstado << "E11"
 7:        EstInicialE3:TypEstado << "E32"
 8:        EstInicial:  TypEstado << EstInicialE1;
 9:      event types
10:        ev1;
11:        ev2;
12:        ev3;
13:        ev4;
14:        ev5;
15:        ev6;
16:        ev7;
17:      exception types
18:        ERRO {msg : String;};
19:      attributes
20:        object
```

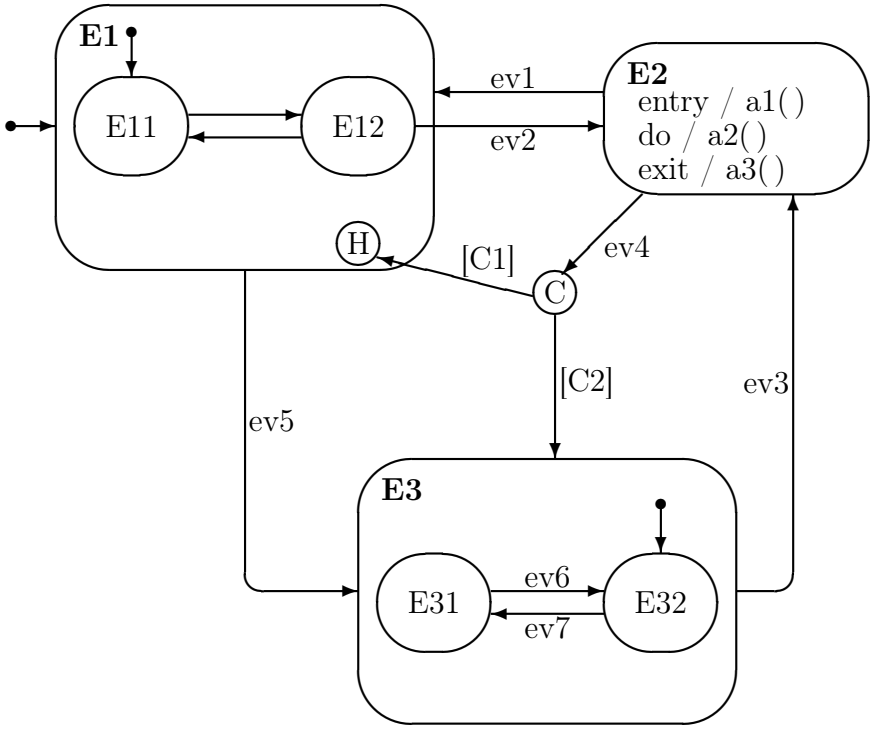


Figura 6.14: State-chart.

```
21:          varHistoriaE1:TypEstado << "E12"
22:
23:       // varEstado indica qual o estado actual
24:          varEstado : TypEstado;
25:
26:       // prox indica qual o estado seguinte. Quando prox é diferente de
27:       // varEstado, então há que provocar uma transição de estado
28:          prox      : TypEstado;
29:
30:       // aborta indica que um evento assíncrono foi sentido, enquanto o estado
31:       // actual estava a processar a respectiva actividade (fimActiv = FALSE)
32:          aborta : Bool;
33:
34:       // fimActiv indica que um estado terminou a sua actividade mas que
35:       // ainda não transitou (transição automática ou espera de evento)
36:       // Permite determinar se, aquando dum evento assíncrono, é necessário
37:       // interromper ou não a actividade do estado actual
38:          fimActiv  : Bool;
39:
40:    operations
41:      class
42:        *cria;  // cria instancia
43:
44:    body
45:      invariants
46:        EstadoValido:
47:          varEstado = "E11" OR varEstado = "E12" OR
48:          varEstado = "E2"  OR varEstado = "E31" OR
49:          varEstado = "E32"
50:            exception ERRO(msg << "Estado Invalido");
51:      methods
52:        object
53:
54:      // -- Escalonador ---------------------------------------
55:
56:          !escalona is
57:            enabling TRUE;  // assume-se que um relógio regula a execução desta operação
58:          {
```

```
59:            // os eventos assíncronos alteram prox
60:            pre (fimActiv = TRUE)
61:              call self.transAutomatica ();  // altera prox
62:            set fimActiv << FALSE;  // eventos duram apenas um ciclo
63:            pre (prox <> varEstado)
64:              call self.mudaEstado();
65:          } end
66:
67:      // -- transAutomatica -----
68:
69:          transAutomatica
70:            method E11 is
71:              enabling varEstado = "E11"
72:            {
73:              set prox << "E12";
74:            } end
75:
76:          transAutomatica
77:            method E12 is
78:              enabling varEstado = "E12"
79:            {
80:              set prox << "E11";
81:            } end
82:
83:      // -- mudaEstado ----------
84:
85:          mudaEstado
86:            method E11 is
87:              enabling prox = "E11"
88:            {
89:              set aborta << FALSE;
90:              call self.estadoE11();
91:            } end
92:
93:          mudaEstado
94:            method E12 is
95:              enabling prox = "E12"
96:            {
97:              set aborta << FALSE;
98:              call self.estadoE12();
99:            } end
100:
101:         mudaEstado
102:           method E2 is
103:             enabling prox = "E2"
104:           {
105:             set aborta << FALSE;
106:             call self.estadoE2();
107:           } end
108:
109:         mudaEstado
110:           method E31 is
111:             enabling prox = "E31"
112:           {
113:             set aborta << FALSE;
114:             call self.estadoE31();
115:           } end
116:
117:         mudaEstado
118:           method E32 is
119:             enabling prox = "E32"
120:           {
121:             set aborta << FALSE;
122:             call self.estadoE32();
123:           } end
124:
125:     // -- Métodos para Tratamento de Eventos ----------------
126:
127:         ^trataEv1 trigger (ev:ev1)
128:           method E2 is
129:             enabling varEstado = "E2"
130:           {
131:             pre (fimActiv = FALSE)
```

```
132:               set aborta << TRUE;
133:             call self.a3();  // executa acção exit de E2
134:             set prox << varHistoriaE1;
135:           } end
136:
137:         ^trataEv2 trigger (ev:ev2)
138:           method E12 is
139:             enabling varEstado = "E12"
140:           {
141:             pre (fimActiv = FALSE)
142:               set aborta << TRUE;
143:             set prox << "E2";
144:           } end
145:
146:         ^trataEv3 trigger (ev:ev3)
147:           method E31 is
148:             enabling varEstado = "E31"
149:           {
150:             pre (fimActiv = FALSE)
151:               set aborta << TRUE;
152:             set prox << "E2";
153:           } end
154:
155:         ^trataEv3 trigger (ev:ev3)
156:           method E32 is
157:             enabling varEstado = "E32"
158:           {
159:             pre (fimActiv = FALSE)
160:               set aborta << TRUE;
161:             set prox << "E2";
162:           } end
163:
164:         ^trataEv4 trigger (ev:ev4)
165:           method E2C1 is
166:             enabling varEstado = "E2" and C1
167:           {
168:             pre (fimActiv = FALSE)
169:               set aborta << TRUE;
170:             call self.a3();  // executa acção exit de E2
171:             set prox << varHistoriaE1;
172:           } end
173:
174:         ^trataEv4 trigger (ev:ev4)
175:           method E2C2 is
176:             enabling varEstado = "E2" and C2
177:           {
178:             pre (fimActiv = FALSE)
179:               set aborta << TRUE;
180:             call self.a3();  // executa acção exit de E2
181:             set prox << EstInicialE3;
182:           } end
183:
184:         ^trataEv5 trigger (ev:ev5)
185:           method E11 is
186:             enabling varEstado = "E11"
187:           {
188:             pre (fimActiv = FALSE)
189:               set aborta << TRUE;
190:             set prox << EstInicialE3;
191:           } end
192:
193:         ^trataEv5 trigger (ev:ev5)
194:           method E12 is
195:             enabling varEstado = "E12"
196:           {
197:             pre (fimActiv = FALSE)
198:               set aborta << TRUE;
199:             set prox << EstInicialE3;
200:           } end
201:
202:         ^trataEv6 trigger (ev:ev6)
203:           method E31 is
204:             enabling varEstado = "E31"
```

```
205:            {
206:              pre (fimActiv = FALSE)
207:                set aborta << TRUE;
208:              set prox << "E32";
209:            } end
210:
211:          ^trataEv7 trigger (ev:ev7)
212:            method E32 is
213:              enabling varEstado = "E32"
214:            {
215:              pre (fimActiv = FALSE)
216:                set aborta << TRUE;
217:              set prox << "E31";
218:            } end
219:
220:      // -- Métodos para Estados -----------------------------
221:
222:          estadoE11 is
223:          {
224:            set varEstado << "E11";
225:            set varHistorialE1 << "E11";
226:            set fimActiv << TRUE;
227:          } end
228:
229:          estadoE12 is
230:          {
231:            set varEstado << "E12";
232:            set varHistorialE1 << "E12";
233:            set fimActiv << TRUE;
234:          } end
235:
236:          estadoE2 is
237:          {
238:            set varEstado << "E2";
239:            call self.a1();  // executa acção entry de E2
240:            { assert (aborta = FALSE);
241:              call self.a2(); }  // executa actividade de E2
242:            set fimActiv << TRUE;
243:          } end
244:
245:          estadoE31 is
246:          {
247:            set varEstado << "E31";
248:            set fimActiv << TRUE;
249:          } end
250:
251:          estadoE32 is
252:          {
253:            set varEstado << "E32";
254:            set fimActiv << TRUE;
255:          } end
256:
257:      // -- Métodos para Actividades -------------------------
258:
259:          a1 is
260:          {
261:            // terminar
262:          } end
263:
264:          a2 is
265:          {
266:            // terminar
267:          } end
268:
269:          a3 is
270:          {
271:            // terminar
272:          } end
273:
274:      // -- Tratamento de Excepções --------------------------
275:
276:      // -- Erro ----------------
277:
```

```
278:          ^ErroExc trigger (exc:ERRO)
279:          {
280:            set varEstado << EstInicial;
281:          } end
282:
283:        class
284:
285:      // -- cria ---------------------------------------------
286:
287:          *cria is
288:          {
289:            set varEstado << "Inicial";
290:            set prox      << EstInicial;
291:            set aborta    << FALSE;
292:            set fimActiv  << FALSE;
293:          } end
294:
295:  end class
```

6.4 Herança de código

A estratégia para codificação do state-chart foi idealizada de modo a considerar as relações hierárquicas entre classes e facilitar a sua repercussão no código. Quando o state-chart duma classe é uma especialização do state-chart da sua superclasse, o facto do código estar dividido por vários métodos facilita a herança das partes comuns às duas classes, a localização dos métodos que devem ser alterados e a identificação de quais os novos métodos a introduzir.

Para cada uma das 6 regras que, em MIDAS, podem ser aplicadas, de forma a que um state-chart especificado numa classe possa ser adaptado ao comportamento duma sua subclasse (subsecção 5.6.5), apresentam-se exemplos que pretendem ilustrar as alterações que se verificam ao nível das classes, tendo em conta que o comportamento do state-chart é conseguido pela execução de métodos no respectivo objecto.

Como exemplo para a presente secção, retoma-se a classe AparelhoCD apresentada na fig. 6.3(a), que se reproduz na fig. 6.15, com ligeiras modificações. Assim, para ilustrar cada uma das regras, apresenta-se uma nova subclasse de AparelhoCD e o respectivo state-chart, permitindo assim visualizar facilmente quais as alterações que se verificam ao nível da classe e do state-chart, devido ao mecanismo de herança.

6.4.1 Conservação do state-chart

A fig. 6.16 ilustra a regra h1, em que, na subclasse AparelhoCD1, são adicionados o atributo preço e a operação activarFiltro que não têm dependência do estado do objecto e que, portanto, não são referidos no state-chart.

6.4.2 Redefinição das actividades e acções dum estado

A aplicação da regra h2 é considerada na fig. 6.17, em que, no state-chart da subclasse AparelhoCD2, se altera a actividade associada ao estado Pronto. No exemplo, a actividade mostraInfoCD é entendida como uma especialização da actividade mostraTítulo, devendo, portanto, ser escrita na classe AparelhoCD2. Seria igualmente possível, no âmbito desta regra, ter apresentado exemplos, onde se efectuariam alterações nas acções (do tipo *entry* e *exit*) associadas a um estado.

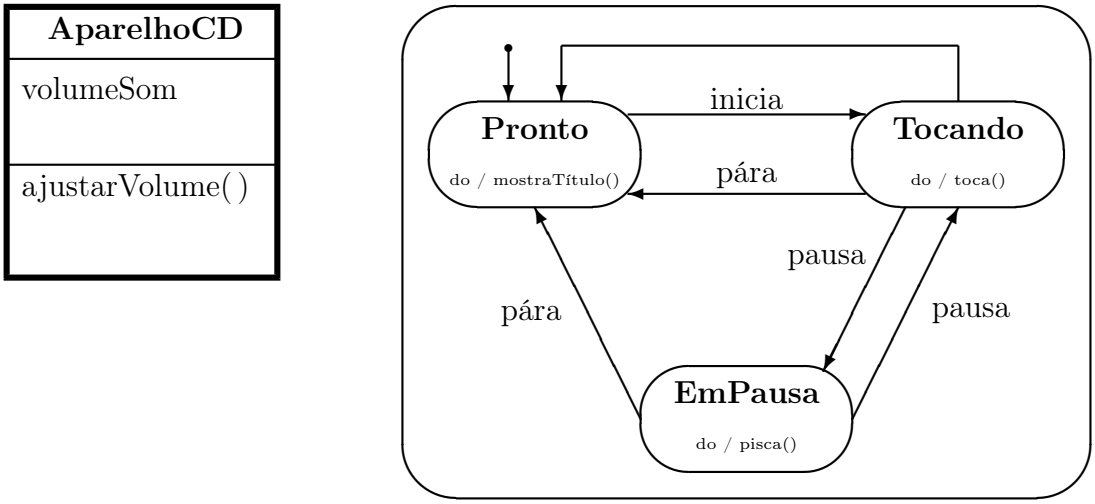


Figura 6.15: A classe AparelhoCD e o respectivo state-chart.

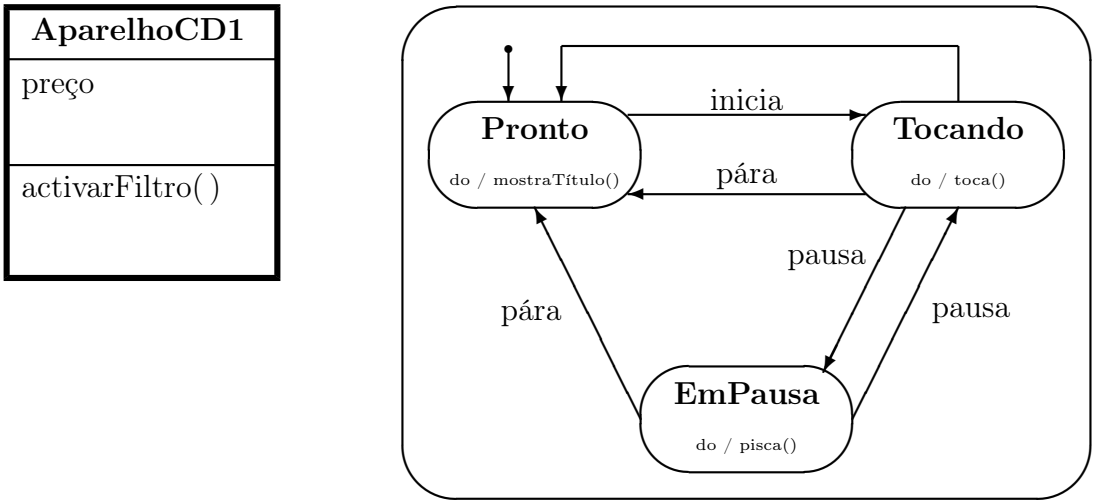


Figura 6.16: State-chart inalterado.

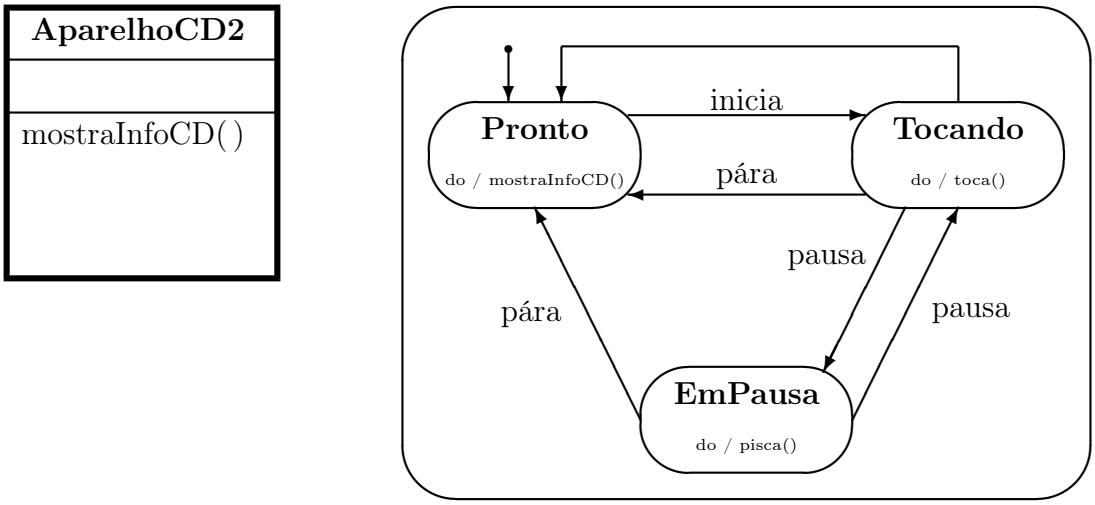


Figura 6.17: State-chart em que se alteraram as actividades dos estados Pronto e Tocando.

6.4.3 Adição de transições e estados

A fig. 6.18 mostra a utilização da regra h3, em que, no state-chart da subclasse AparelhoCD3a, se introduz uma transição nova, sensível a um evento herdado (o evento inicia já está disponível no state-chart da superclasse). Este facto implica a reescrita na subclasse do método inicia que trata esse evento. É também possível introduzir novas transições sensíveis a eventos novos. A fig. 6.19 pretende ilustrar essa situação com a introdução duma transição sensível a um novo evento (vai).

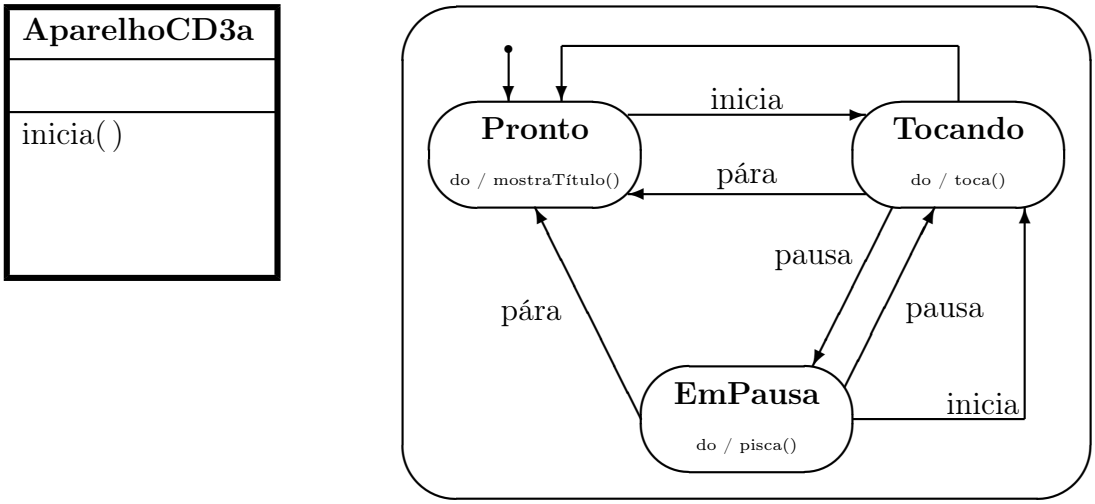


Figura 6.18: State-chart em que se acrescentou uma nova transição.

Como a fig. 6.19 mostra, esta regra pressupõe igualmente que se introduza, no state-chart da subclasse AparelhoCD3b, um novo estado Mudando, que permite alterar a pista que está actualmente a ser reproduzida. Este exemplo mostra que a introdução dum novo estado (Mudando) implica obrigatoriamente a introdução de transições (no caso, sensíveis a novos eventos) que liguem esse novo estado a alguns dos estados herdados. Na nova subclasse, será necessário, aquando do processo de geração de código, introduzir novos métodos: para o evento vai, para o estado Mudando e para a acção *entry* muda.

Ainda no âmbito da regra h3, é possível refinar um estado, incluindo no seu interior outros sub-estados. Esta possibilidade encontra-se ilustrada na fig. 6.20, em que o estado Tocando contém dois sub-estados Normal e Intro. Considerou-se, neste caso, uma nova funcionalidade (tocaIntro) que permite ouvir os primeiros 15s de cada pista do CD. Esta funcionalidade do sistema é iniciada quando for gerado o evento intro.

6.4.4 Alteração do estado destino duma transição

No state-chart, apresentado na fig. 6.21, da subclasse AparelhoCD4 introduziu-se um novo estado Lendo, que permite ler uma dada programação para o CD (selecção dum subconjunto das pistas a reproduzir). Este exemplo concretiza o uso da regra h4, em que o novo estado introduzido (Lendo) é colocado no meio de 2 estados herdados (Pronto e Tocando), o que implica a alteração do estado destino da transição que tem associado o evento inicia.

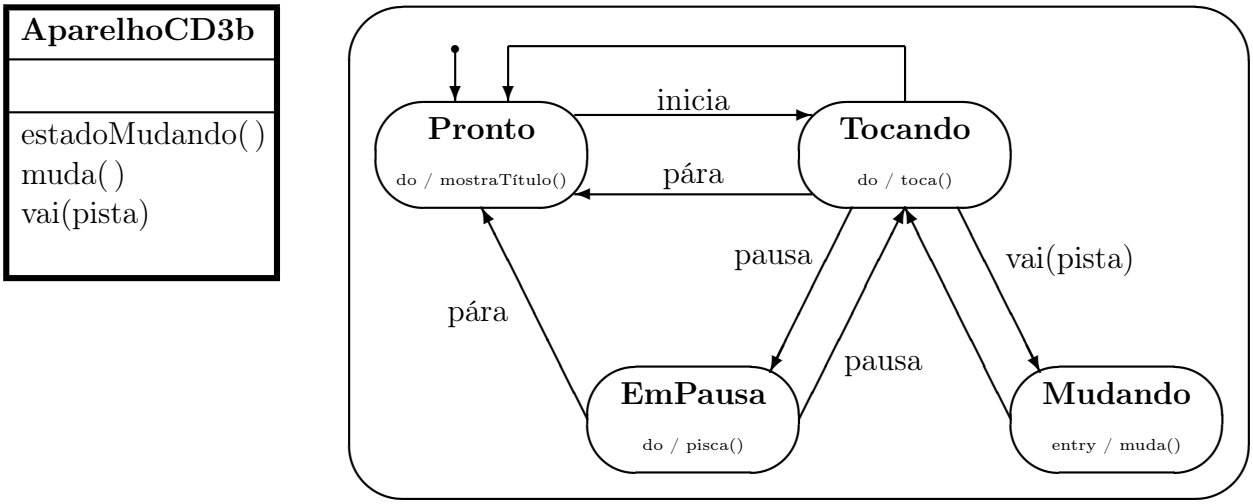


Figura 6.19: State-chart em que se acrescentaram o estado Mudando, 2 transições, o evento vai e a actividade muda.

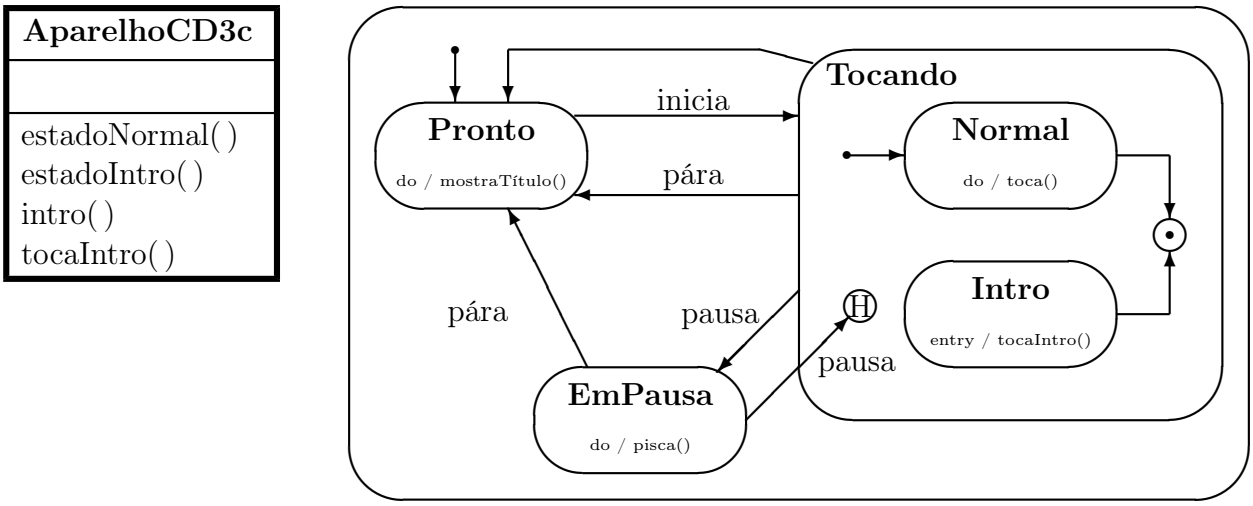


Figura 6.20: State-chart em que se refinou o estado Tocando.

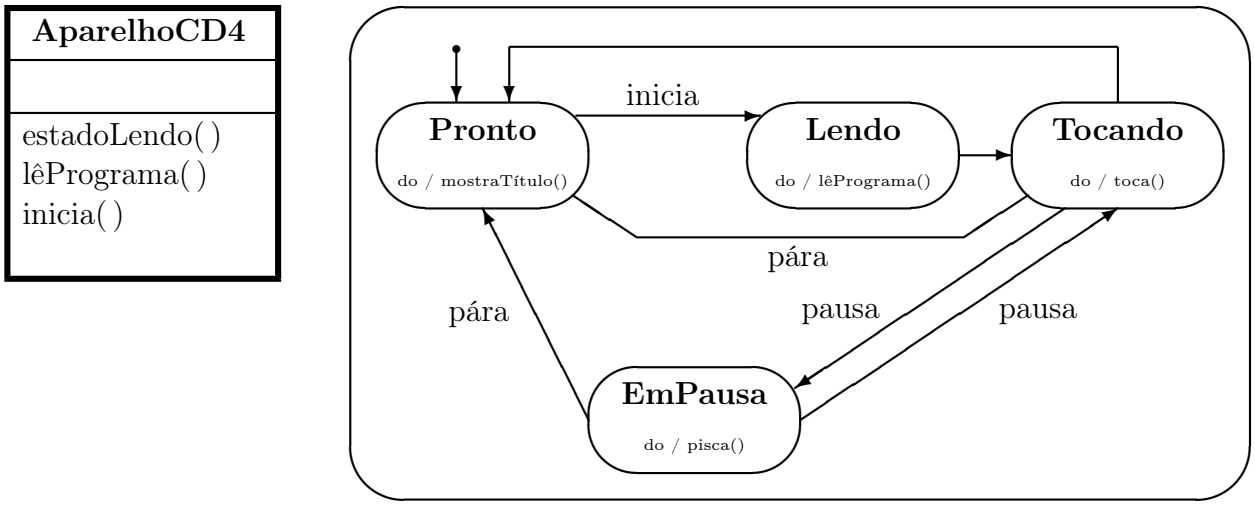


Figura 6.21: State-chart em que se acrescentaram o estado Mudando e a respectiva actividade lêPrograma, entre 2 estados herdados.

6.4.5 Remoção de transições

A regra h5 é exemplificada na fig. 6.22, em que, no state-chart da subclasse AparelhoCD5, se removeu a transição que ligava os estados EmPausa e Pronto. Neste caso, não é possível parar directamente o CD, caso este esteja em pausa. Esta remoção está de acordo com os pressupostos da regra, pois no state-chart da classe AparelhoCD5 continua a existir uma referência ao evento pára numa das transições que liga os estados Tocando e Pronto.

Relativamente aos métodos disponíveis na superclasse, é necessário, na subclasse AparelhoCD5, reescrever o método respeitante ao evento pára, pois esse evento tem de ser ignorado quando o estado actual do state-chart for EmPausa.

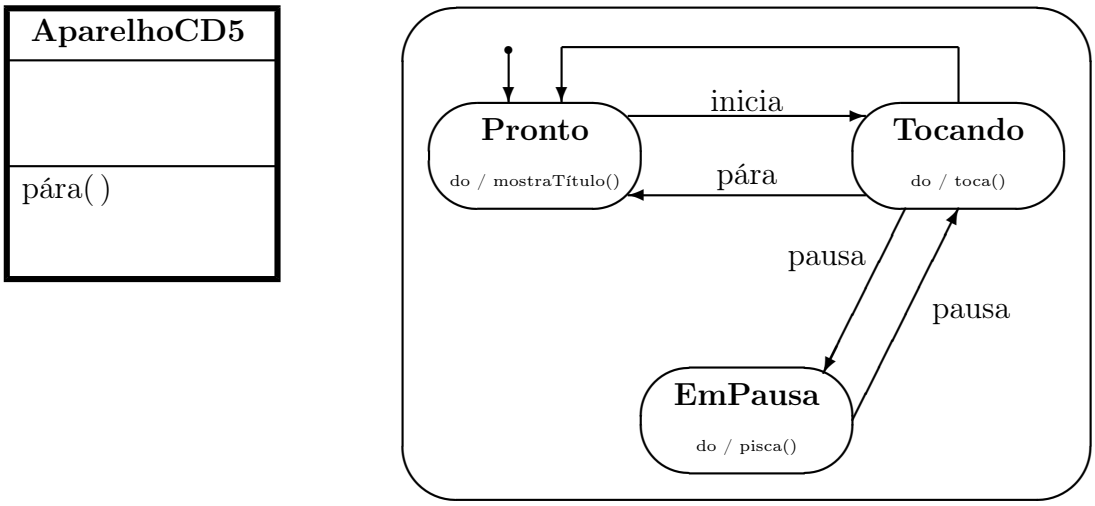


Figura 6.22: State-chart em que se removeu a transição, entre 2 estados herdados (Pronto e EmPausa), sensível ao evento pára.

6.4.6 Especialização de etiquetas de transições

A aplicação da regra h6 é retratada na fig. 6.23, em que, no state-chart da subclasse AparelhoCD6, se altera a etiqueta da transição que liga o estado Pronto ao estado Tocando. No exemplo, é adicionada uma condição [prontoCD] que refina a forma como a transição pode ser disparada, pelo que se deve adicionar o atributo prontoCD à classe AparelhoCD6. Seria também possível, no âmbito desta regra, ter apresentado exemplos, onde se acrescentariam acções à transição.

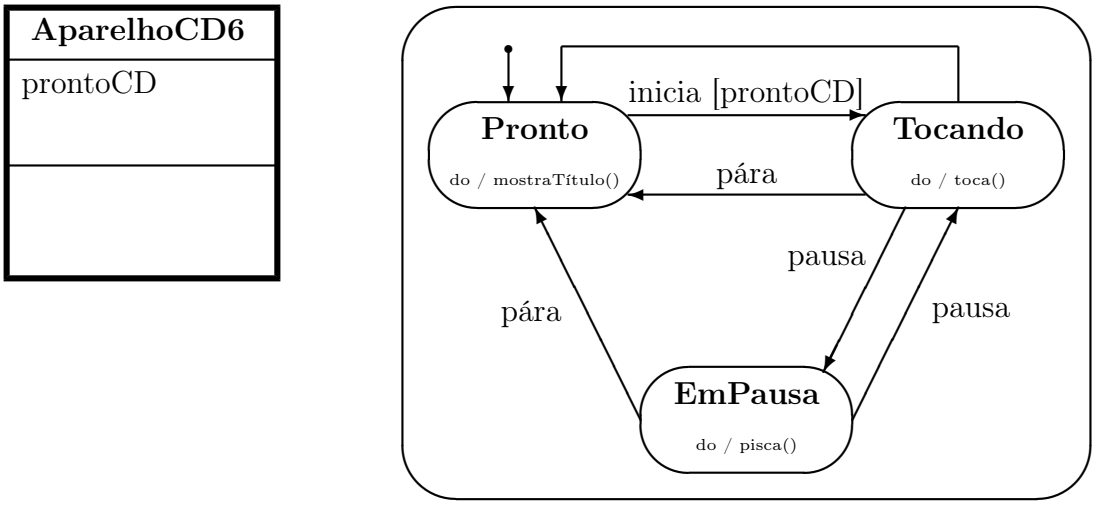


Figura 6.23: State-chart em que se refinou a transição, entre os estados herdados (Pronto e Tocando), pela adição duma condição à etiqueta.

6.5 Resumo final

Este capítulo tratou o uso da linguagem Oblog como representação unificada para sistemas embebidos. Foram indicadas algumas recomendações que permitem transformar os diagramas UML (diagramas de classes, objectos e state-charts) no código Oblog.

Foi dada forte atenção à transformação de state-charts para código, por ser essa a parte mais complexa de todo o processo de codificação, pois o meta-modelo state-chart apresenta muitos mecanismos de modelação. A apresentação da tradução fez-se com recurso a casos típicos de modelação, nomeadamente transições iniciais, acções, actividades, transições a vários níveis de profundidade, transições automáticas, transições condicionais, transições de grupo, conectores história, e state-charts com hierarquia.

Fez-se igualmente um estudo sobre a forma como as relações hierárquicas entre classes se reflectem no código Oblog, tomando em consideração as 6 regras de herança entre state-charts que podem aplicar-se no âmbito da metodologia Midas (conservação do state-chart, redefinição das actividades e acções dum estado, adição de transições e estados, alteração do estado destino duma transição, remoção de transições e especialização de etiquetas de transições).

Capítulo 7

Validação da Metodologia Proposta

Quem não experimenta não sabe.
Mais vale o exemplo que a doutrina.

Sumário

Neste capítulo, pretende mostrar-se a utilização efectiva da metodologia MIDAS *a casos práticos, com o intuito de validar a sua adequabilidade à modelação de sistemas embebidos. Como exemplos práticos de estudo, apresentam-se dois sistemas: o Sistema de Supervisão de Iluminação (SSI), desenvolvido para supervisionar a iluminação exterior dum edifício, e o Sistema de Controlo das Linhas* HIDRO *(SCLH), responsável pela coordenação dum conjunto de linhas de produção de auto-rádios.*

Índice

Neste capítulo, pretende validar-se a aplicação da metodologia MIDAS a casos práticos, pondo em evidência algumas questões de modelação que foram introduzidas nos capítulos anteriores. Como exemplos práticos de estudo, são considerados dois sistemas: o Sistema de Supervisão de Iluminação (SSI) e o Sistema de Controlo das Linhas HIDRO (SCLH). Estes dois exemplos são sistemas reais (e não exemplos académicos escolhidos pelo autor) que foram desenvolvidos num ambiente externo à instituição e com a colaboração dos clientes.

O SSI permite supervisionar a iluminação exterior dum edifício, com o principal objectivo de detectar a ocorrência de avarias nos pontos de luz. Trata-se dum exemplo relativamente pequeno (de pouca complexidade), introduzido com o intuito principal de mostrar as técnicas genéricas para análise e modelação de sistemas no âmbito da metodologia MIDAS.

O SCLH é responsável pela coordenação dum conjunto de linhas de produção de auto-rádios, sendo um sistema de muito maior complexidade que o SSI. Admitindo que o primeiro exemplo esclarece as questões principais, é possível com o SCLH aprofundar esses conhecimentos e introduzir novas questões que não fazia sentido abordar anteriormente, devido à dimensão do SSI.

Nas secções seguintes, descreve-se a forma como esses dois sistemas foram desenvolvidos, apresentando os diversos diagramas e documentos que resultaram desse processo, bem como alguns comentários e reflexões sobre a aplicação da metodologia.

7.1 Sistema de supervisão de iluminação

7.1.1 O cliente

O Sistema de Supervisão de Iluminação (SSI) foi desenvolvido por solicitação do Banco de Portugal, cujo edifício sede está localizado na Rua do Ouro em Lisboa.

7.1.2 O problema

Verificou-se a necessidade de instalar um sistema capaz de fazer a supervisão da iluminação exterior (decorativa) do edifício, com o principal objectivo de detectar a ocorrência de avarias nos pontos de luz. Foram identificadas as seguintes dificuldades:

- Os custos relacionados com a manutenção do sistema eram elevados e verificava-se que os pontos de luz se mantinham avariados, dado que essa situação não era atempadamente detectada.
- Não havia um mecanismo automático que garantisse o deslastre de cargas, nomeadamente a ligação desfasada e individual dum conjunto de pontos de luz.
- A verificação do cumprimento do número de horas de funcionamento das lâmpadas que os fornecedores anunciavam era executada duma forma manual.
- O sistema não integrava nenhuma funcionalidade que possibilitasse programar os tempos de actividade, incluindo detectores de luminosidade natural.

7.1.3 Descrição do ambiente

O SSI será instalado no edifício que serve de sede ao Banco de Portugal. O edifício ocupa um quarteirão inteiro da Baixa de Lisboa, confinando com a Rua do Ouro, a Rua de São Julião, o Largo de São Julião e a Rua do Comércio. O SSI será responsável pela supervisão da iluminação das 5 áreas exteriores do edifício (4 fachadas e o telhado). Usando como referência as plantas do edifício mostradas nas fig. 7.1 e 7.2, os 222 pontos de luz instalados no edifício estão distribuídos da seguinte forma:

- Na fachada da Rua de São Julião, existem 26 pontos de luz no 4^o andar e 33 pontos de luz no 1^o andar (31 projectados para cima para iluminar as paredes e 2 projectados para baixo para iluminar uma entrada).
- Na fachada da Rua do Ouro, estão disponíveis 12 pontos de luz no 4^o andar e 20 pontos de luz no 1^o andar (10 projectados para cima e 10 projectados para baixo).
- Na fachada do Largo de São Julião, estão instalados 6 pontos de luz no rés-do-chão.
- Na fachada da Rua de Comércio estão colocados 20 pontos de luz no 4^o andar e 33 pontos de luz no 1^o andar.
- No telhado existem 72 pontos de luz, dispostos de forma irregular.

Cada um dos pontos de luz já instalados no edifício é composto por uma armadura, uma lâmpada (de 75 ou 150w), um condensador, uma reactância e um ignitor. Este último necessita de ser instalado para permitir que a lâmpada arranque, pois dada a sua tecnologia, só acende quando lhe é aplicada um tensão elevada (2.5kV) aos seus terminais.

O SSI será usado por dois perfis diferenciados de utilizadores: os responsáveis pela instalação eléctrica e os electricistas. Aos primeiros dar-se-ão privilégios de supervisor e aos segundos corresponderá o modo de utilização normal. Espera-se que o SSI tenha uma interface o mais amigável e intuitiva possível com os seus utilizadores.

7.1.4 Descrição das funções disponibilizadas

O objectivo principal que o SSI deve cumprir consiste em identificar os pontos de luz não operacionais, devido, por exemplo, a uma lâmpada fundida ou a um ignitor danificado. Sempre que for detectado algum problema, deve accionar-se o alarme adequado, de modo a que a substituição ou reparação das peças em mau estado se faça o mais rapidamente possível. O alarme deve incluir a origem do problema, a razão e o instante em que ocorreu.

O SSI deve também permitir programar quando e como devem ser ligados os vários pontos de luz. Essa programação deve facilitar a identificação de unidades lógicas (grupos) de iluminação, organizadas segundo a seguinte hierarquia: um ponto de luz individual, um grupo de pontos de luz consecutivos (exemplo, os 10 pontos de luz mais à esquerda do 4^o andar da fachada da rua do Comércio), os pontos de luz dum andar duma fachada, todos os pontos de luz duma dada fachada, ou todo o edifício. É importante que seja estabelecido um mecanismo que permita a ligação de grupos de pontos de luz, sem que ocorra qualquer sobrecarga no quadro eléctrico (por exemplo, ligando os pontos de luz em série ou em pequenos grupos mas desfasadamente).

Devem considerar-se 4 modos distintos para a programação dos pontos de luz, que a seguir se explicam em pormenor:

- Em modo **manual**, o utilizador indica quais os pontos de luz a ligar, permanecendo estes ligados, até ser fornecida uma nova ordem ou até se mudar para um modo automático.

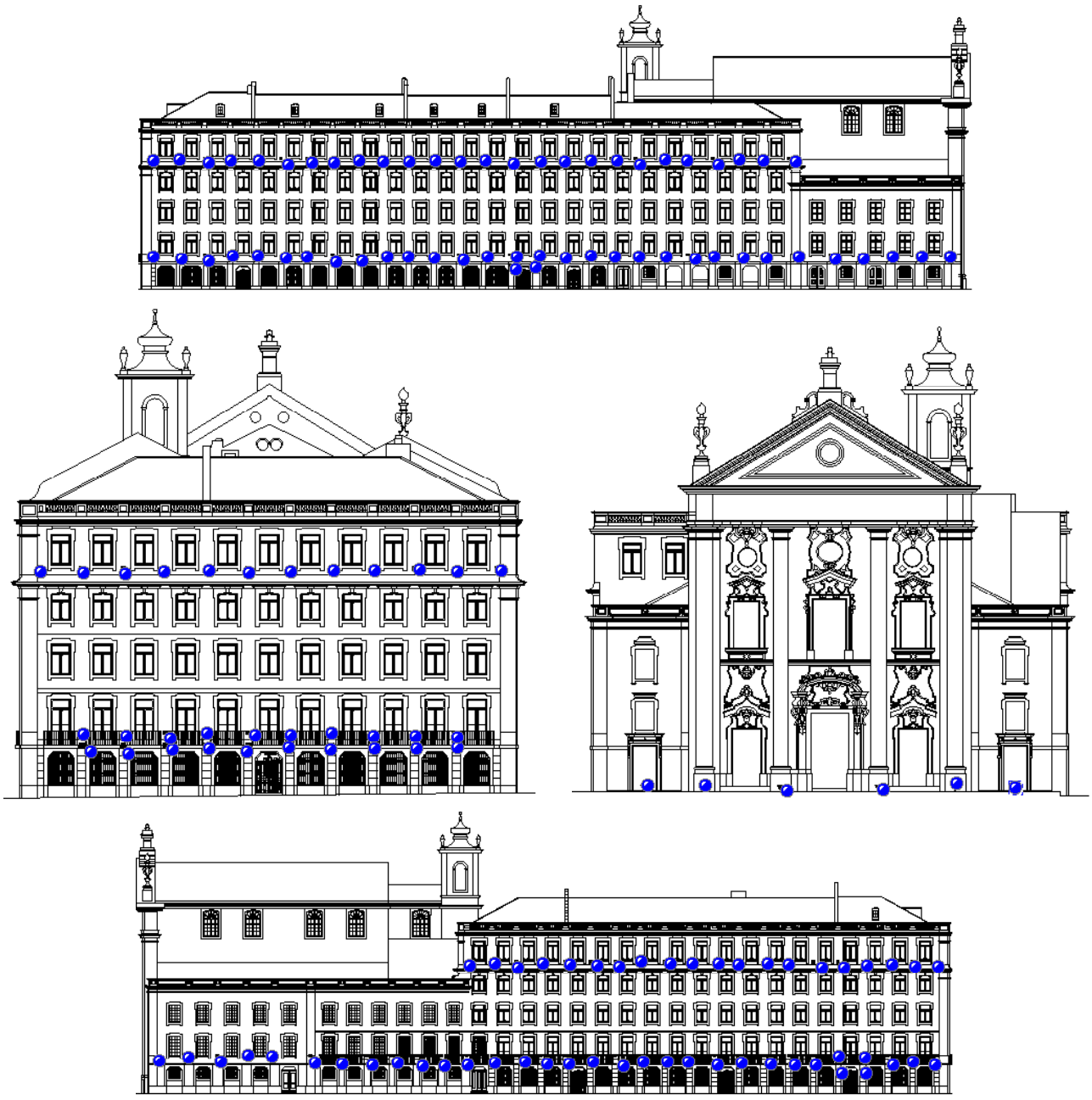


Figura 7.1: Plantas dos alçados do edifício sede do Banco de Portugal. Vistas da Rua de São Julião (em cima), da Rua do Ouro (no centro à esquerda), do Largo de São Julião (no centro à direita) e da Rua do Comércio (em baixo).

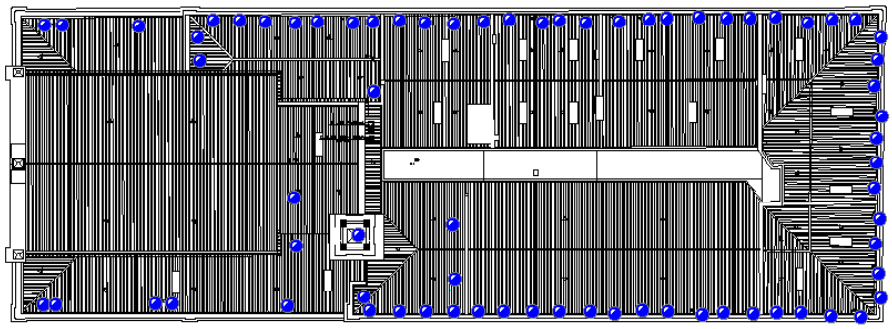


Figura 7.2: Planta do telhado do edifício sede do Banco de Portugal.

- Em modo **automático por foto-célula**, os pontos de luz acendem-se, assim que uma foto-célula, instalada no telhado do edifício, indicar que o nível da luminosidade natural se encontra abaixo dum valor de referência. Por simplicidade de linguagem, quando houver luminosidade abaixo do referencial fixado diz-se que a foto-célula está ligada ou activa e, no caso contrário, diz-se que a foto-célula está desligada ou inactiva. Associada a este modo está, especificada pelo responsável, uma única configuração dos pontos de luz que estão ligados durante a noite e desligados de dia. Este modo de programação permite, por exemplo, que o sistema de iluminação seja ligado mais tarde nos dias de verão do que nos dias de inverno.
- O modo **automático por relógio** possibilita que os pontos de luz sejam ligados de acordo com um horário programável. Devem existir dois tipos de dias: normais e especiais (para incluir feriados, fins-de-semana e outros). Existirá um calendário (organizado por meses) para indicar o tipo de cada um dos dias do mês. Para cada um dos 2 tipos de dia e para cada período contíguo de tempo, devem indicar-se quais os pontos de luz a ligar, considerando-se, como unidade temporal mínima de programação, a hora, razão pela qual não é necessária a indicação dos minutos (nem dos segundos). Por exemplo, pode indicar-se que das 00h00 às 06h00 está todo o edifício iluminado; das 18h00 às 20h00 estão ligados todos os pontos de luz do 4^o andar e do telhado; e das 20h00 às 24h00 volta a iluminar-se todo o edifício. Esta descrição pressupõe implicitamente que, das 06h00 às 18h00, todo o sistema de iluminação está desligado.
- O modo **automático por foto-célula e relógio** significa que a ligação dum dado ponto de luz só se efectua quando a foto-célula está activa "**E**" (no sentido booleano do termo) a programação por relógio indica que esse ponto de luz deve ser ligado. Retomando o exemplo considerado no item anterior, admita-se que até às 21h16 a foto-célula está desligada (valor da luminosidade natural acima do valor de referência); nesta situação, durante o período das 18h00 às 21h16, os pontos de luz estarão desligados. Note-se que apesar do período mínimo de programação ser uma hora, as lâmpadas se acendem, mal as duas condições (relógio + foto-célula) se verifiquem, ou seja, não há que esperar até à próxima hora certa (no caso, 22h00), para verificar se as lâmpadas têm ou não que ser ligadas. Este modo pode ser interpretado como uma extensão ao modo automático por relógio em que a programação do relógio é cumprida assim que a foto-célula está activa

A foto-célula é um dispositivo electrónico não programável (i.e. a intensidade de luz para a qual ela se activa não pode ser alterada por software). Contudo, é possível ajustar a intensidade

de activação da foto-célula por hardware, ou seja, actuando directamente sobre um botão disponível no dispositivo que controla a foto-célula (controlador). Para um funcionamento mais correcto (menos sujeito a interferências), a foto-célula foi colocada no telhado do edifício

Deve também ser feito um inventário, em que se armazenam dados relativos a todos os pontos de luz instalados no edifício. Para cada um deles, deve armazenar-se informação respeitante à armadura (referência, número de substituições), ao condensador (referência, capacidade, número de substituições), à reactância (referência, indutância e número de substituições), ao ignitor (referência e número de substituições) e à lâmpada (referência, fabricante, potência, número de substituições e número de horas em funcionamento).

Existe um aparelho de medida (fig. 7.3), instalado junto ao quadro eléctrico da instalação, que permite captar diversas grandezas eléctricas da alimentação. O SSI, relativamente a esses valores, deve monitorizá-los, armazená-los (criação de *logs*) para posterior geração de relatórios e disponibilizá-los ao utilizador. A monitorização desse conjunto de valores deve ser feita em cada hora, podendo, no entanto, este período ser alterado, caso o responsável o indique expressamente. As grandezas disponibilizadas pelo aparelho de medida são as seguintes: tensão simples, intensidade de corrente, potência activa, potência capacitiva, potência indutiva, factor de potência, tensão composta, frequência e energia. Ainda junto ao quadro eléctrico, existe um conjunto de contactos auxiliares associados aos diversos disjuntores diferenciais, sendo cada um destes responsável pela protecção duma parte do sistema eléctrico. Cada um desses contactos auxiliares permite ao SSI averiguar se o respectivo disjuntor disparou ou não (i.e. se a parte do sistema de iluminação que esse disjuntor protege está ou não a ser alimentada).

Figura 7.3: O aparelho medida existente junto ao quadro eléctrico da instalação.

Aos electricistas (utilizadores normais), apenas é permitido a visualização de alarmes e a impressão de relatórios. Podem ser solicitados relatórios impressos com informação relativa à

ocorrência de alarmes ou ao historial dos valores eléctricos provenientes do aparelho de medida. Todas as tarefas que necessitem de alterar qualquer parâmetro (a informação relativa aos pontos de luz ou a programação da ligação das lâmpadas) são permitidas unicamente aos responsáveis, que também têm acesso, como é óbvio, a todas as operações a que os electricistas têm.

7.1.5 Levantamento de requisitos

As descrições do problema, do ambiente e das funcionalidades pretendidas para o SSI, apresentadas anteriormente, não foram captadas todas na mesma altura. Foi seguida uma estratégia iterativa que contou com a boa colaboração do outro interlocutor (o cliente).

Na reunião inicial, foram colocadas diversas questões sobre o SSI, tendo ficado clara uma grande parte das suas características, dos seus objectivos e do seu funcionamento. Algumas particularidades foram propositadamente proteladas (por exemplo, informação disponibilizada pelo aparelho de medida, formato específico dos relatórios), para não sobrecarregar o levantamento de requisitos na primeira reunião com aspectos demasiado minuciosos. Uma leitura atenta do material recolhido mostrou algumas indefinições, pelo que se seguiram algumas trocas de informação no sentido de as clarificar.

7.1.6 Análise

Diagrama de contexto

A estratégia para criar o diagrama de contexto consiste nos 3 passos seguintes:

1. Construir uma lista de actores, entradas e saídas.
2. Desenhar o diagrama de contexto, escolhendo o tipo adequado para cada uma das ligações entre entidades externas e o sistema.
3. Especificar pormenorizadamente as ligações numa tabela.

Os actores mais facilmente identificados em qualquer sistema são os humanos que com ele têm de interagir. Para o SSI, apenas se prevê a interacção com os responsáveis e os electricistas.

Uma leitura atenta do documento que contém a descrição do SSI, permite identificar claramente os pontos de luz como um actor externo ao sistema, pois o objectivo principal do SSI é precisamente detectar avarias nos pontos de luz instalados. Por outro lado, a foto-célula, o aparelho de medida e os contactos dos disjuntores também foram identificados como actores. O historial, por ser disponibilizado aos utilizadores, foi considerado um actor, embora também pudesse ser considerado um objecto interno ao sistema, uma vez que, como mais à frente se verá, existe funcionalidade associada aos utilizadores para disponibilizar essa informação. A opção tomada, por não colidir com o pretendido, resultou da possibilidade de, futuramente, se pretender que o historial seja gerado automaticamente para o exterior (i.e. sem a necessidade dum pedido expresso proveniente dum utilizador).

A lista de actores, entradas e saídas relativa ao SSI encontra-se descrita no quadro 7.1.

A fig. 7.4 mostra uma versão, não pormenorizada, do diagrama de contexto para o SSI. Nele podem ver-se os actores que interactuam com o SSI.

A fig. 7.5 mostra o diagrama de contexto para o SSI, onde, além dos actores, se podem observar as mensagens (e respectivos tipos) que circulam entre o SSI e os actores.

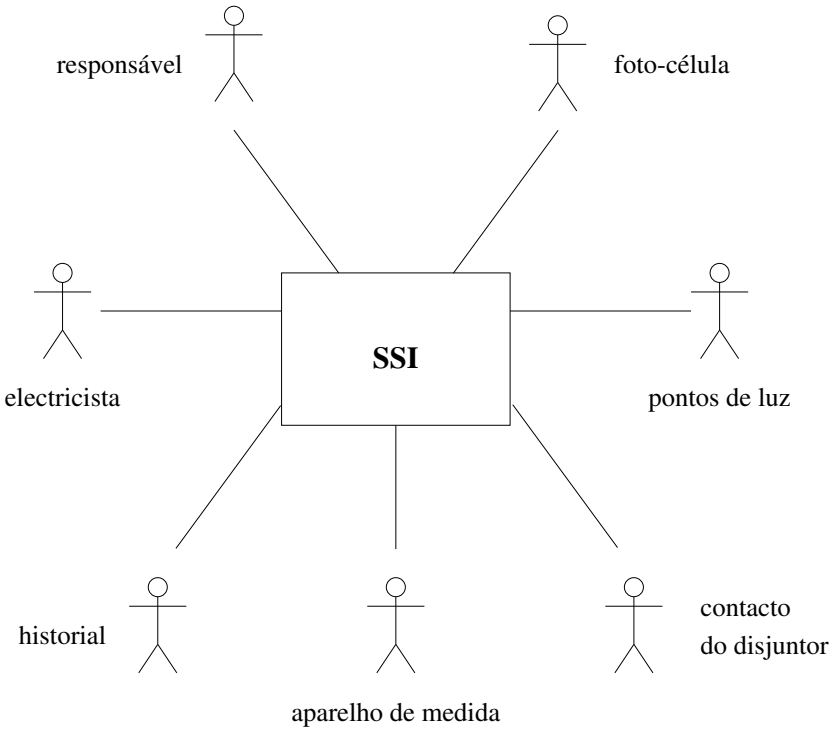


Figura 7.4: O diagrama de contexto do SSI (versão inicial).

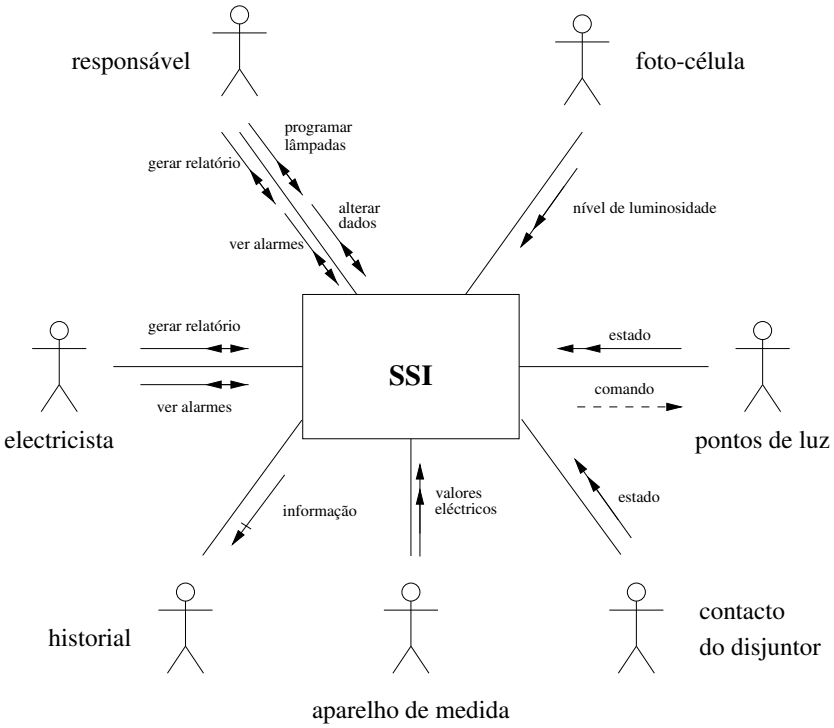


Figura 7.5: O diagrama de contexto do SSI (versão final).

Actores	Entradas	Saídas
Responsável	Valores eléctricos	Actuar pontos de luz
Electricista	Nível de luminosidade	Informação historial
Pontos de luz	Estado dos pontos de luz	
Foto-célula	Programar pontos de luz	
Aparelho de medida	Imprimir relatórios	
Historial	Alterar dados	
Contacto do disjuntor	Ver alarmes	
	Estado do contacto	

Tabela 7.1: Lista de actores, entradas e saídas do SSI.

Para terminar o desenho do diagrama de contexto do SSI, procede-se à pormenorização das ligações entre o sistema e os actores, informação essa que será uma referência importante para todas as fases subsequentes do processo de desenvolvimento (consultar quadro 7.2). Não se propõe nenhuma sintaxe fixa para a definição dessa informação, embora em projectos que envolvam muitas pessoas seja provavelmente vantajoso a imposição dum dado formato. Note-se que, para informação muito elaborada (exemplo, a informação que estabelece o programa de ligação dos pontos de luz), apenas é apresentado um comentário que tenta descrever sucintamente o tipo de informação esperada para essa conexão. Os símbolos '|' e '+' significam, respectivamente, alternativa e reunião.

Ligação	Definição
Valores eléctricos	Tensão + Corrente + Potência Activa + Potência Capacitiva + Potência Indutiva + Factor de potência + Tensão Composta + Frequência + Energia
Nível de luminosidade	Dia | Noite
Estado do ponto de luz	Ligado | Desligado | Avariado
Programar pontos de luz	"Alteração ou definição, pelo supervisor, do programa a usar para actuar sobre os pontos de luz"
Impressão relatórios	"Pedido, pelo utilizador, dum relatório com as ocorrências verificadas no sistema"
Alterar dados	"Introdução, pelo supervisor, de dados relativos aos pontos de luz"
Ver alarmes	"Pedido, pelo utilizador, dos alarmes gerados no sistema"
Estado do contacto	Disparado | Não disparado
Actuar pontos de luz	Ligar | Desligar
Informação historial	Tensão + Corrente + Potência Activa + Potência Capacitiva + Potência Indutiva + Factor de potência + Tensão Composta + Frequência + Energia

Tabela 7.2: Definição das ligações externas do SSI.

Diagrama de casos de uso

Com base nos actores identificados no diagrama de contexto, podem identificar-se os seguintes casos de uso:

1. Registar alterações de dados.
2. Mostrar alarmes.
3. Gerar relatório.
4. Gerar informação de historial.
5. Monitorizar pontos de luz.
6. Programar ligação dos pontos de luz.

A utilização de valores etiquetados, no caso específico {ref=n}, permite numerar cada caso de uso, o que facilita a forma de relacionar os vários modelos (ou diagramas) que vão sendo criados ao longo do projecto.

Identificados que estão os casos de uso, apresenta-se uma descrição sucinta de cada um deles no quadro 7.3. Tomando como base essas descrições textuais, pode desenhar-se o diagrama de casos de uso do SSI (fig. 7.6), em que se mostram quais os actores envolvidos em cada um dos casos de uso.

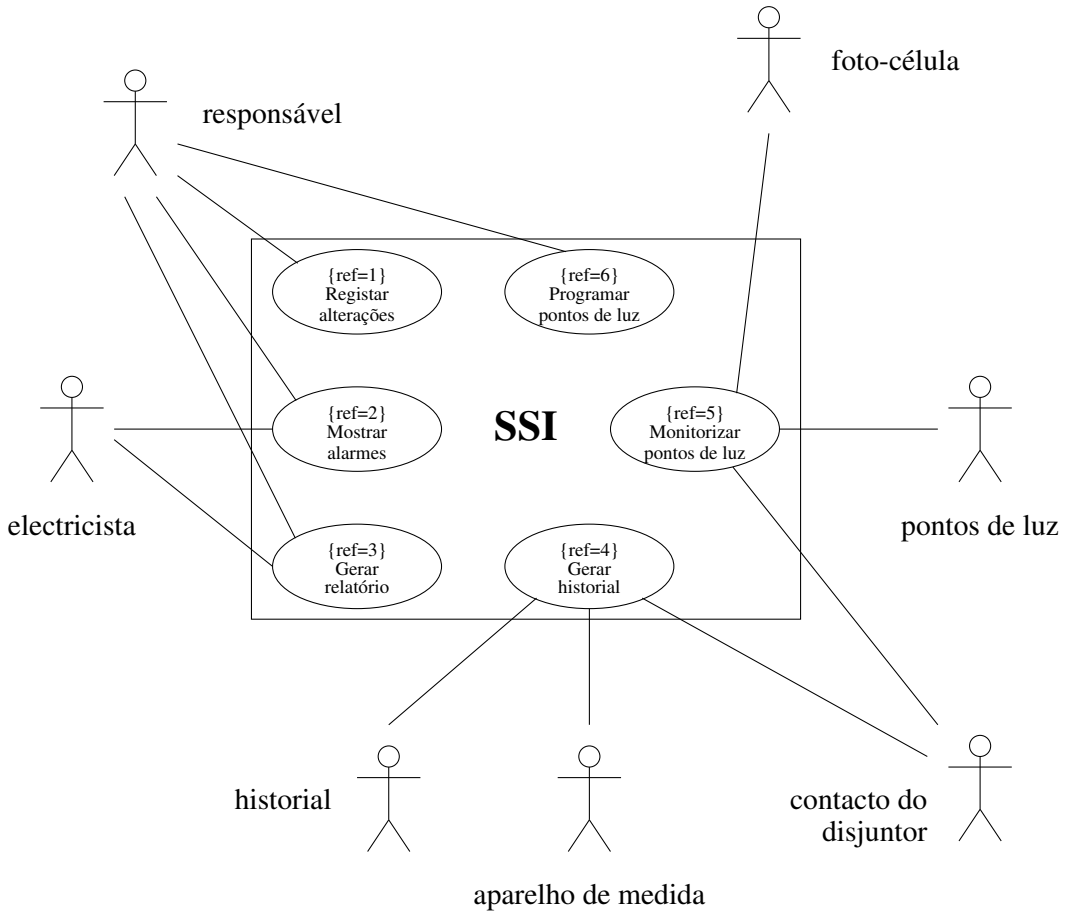


Figura 7.6: O diagrama de casos de uso do SSI.

Neste exemplo, não se torna relevante a utilização das relações entre casos de uso (≪extends≫ e ≪uses≫), embora se reafirme aqui a sua utilidade em outros casos práticos. Também não

1. Registar alterações É iniciado pelo responsável para alterar os dados relativos aos pontos de luz que estão armazenados no sistema. Quando, por exemplo, uma lâmpada avariada é trocada por uma nova, o responsável deve registar essa alteração.
2. Mostrar alarmes É usado pelo utilizador (responsável ou electricista) para verificar se há alguma avaria nos pontos de luz.
3. Gerar relatório É iniciado pelo utilizador (responsável ou electricista) para imprimir as ocorrências de situações anómalas que se verificaram no sistema. Inclui avarias de pontos de luz e valores não esperados para as grandezas eléctricas devolvidas pelo aparelho de medida.
4. Gerar historial É um processo que regista, com um carácter periódico, as grandezas eléctricas fornecidas pelo aparelho de medida, para posterior processamento.
5. Monitorizar pontos de luz É um processo continuamente em execução que, mediante o programa de ligação activo, liga e desliga os pontos de luz. É também responsável pela detecção de pontos de luz avariados, facto que deve registar através da geração dum alarme, para posteriormente os utilizadores tomarem conhecimento do facto.
6. Programar pontos de luz É utilizado pelo responsável para definir o programa a usar para a ligação dos pontos de luz. O programa pode ser colocado num de 4 modos de controlo: manual; automático por foto-célula; automático por relógio; ou automático por foto-célula e relógio.

Tabela 7.3: Descrição sumária dos casos de uso do SSI.

se antevê a necessidade de usar os cenários como mecanismo de modelação, dado que, para o SSI, os casos de uso são todos simples (pouco elaborados). Contudo, apresenta-se, na fig. 7.7, a descrição dum cenário relativo ao caso de uso "Monitorizar pontos de luz", empregando para o efeito um diagrama de sequência.

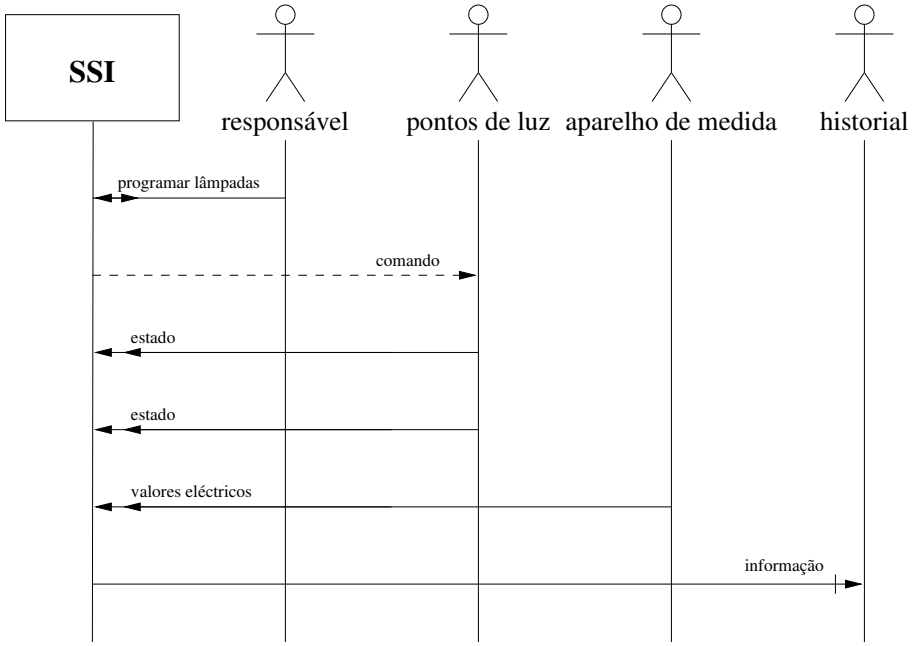


Figura 7.7: Um diagrama de sequência para um cenário relativo ao caso de uso "Monitorizar pontos de luz".

Diagrama de objectos

Para cada caso de uso, fez-se um levantamento de quais as dimensões do espaço de análise que predominantemente cobriam. O quadro 7.4 faz um resumo desse levantamento, mostrando as categorias de objectos a que cada caso de uso dá origem. Recorde-se que os objectos-entidade, objectos-interface e objectos-função cobrem essencialmente as dimensões informação, apresentação e comportamento, respectivamente.

Caso de uso	interface	entidade	função
1. Registar alterações	X	X	
2. Mostrar alarmes	X		
3. Gerar relatório	X		X
4. Gerar historial	X	X	X
5. Monitorizar pontos de luz	X	X	X
6. Programar pontos de luz	X	X	

Tabela 7.4: As categorias dos objectos para cada um dos casos de uso do SSI.

Através da informação que consta do quadro 7.4, foi possível obter o diagrama de objectos do SSI representado na fig. 7.8, onde se podem observar:

- 9 objectos-interface — Um para cada actor, sendo que o responsável e o electricista partilham o mesmo objecto-interface, que pode ser visto como composto por 4 objectos-interface (um para cada interacção possível entre o responsável e o sistema).
- 3 objectos-entidade — Um para guardar os programas de ligação dos pontos de luz; outro para guardar as informações relativas aos pontos de luz; e finalmente um outro para o historial.
- 4 objectos-função — Um activo para fazer a supervisão dos pontos de luz; outro para gerar informação de historial; e um outro para gerar relatórios. Existe igualmente um objecto relógio (da sub-categoria ≪timer≫) responsável por fornecer informação temporal aos outros objectos da comunidade.

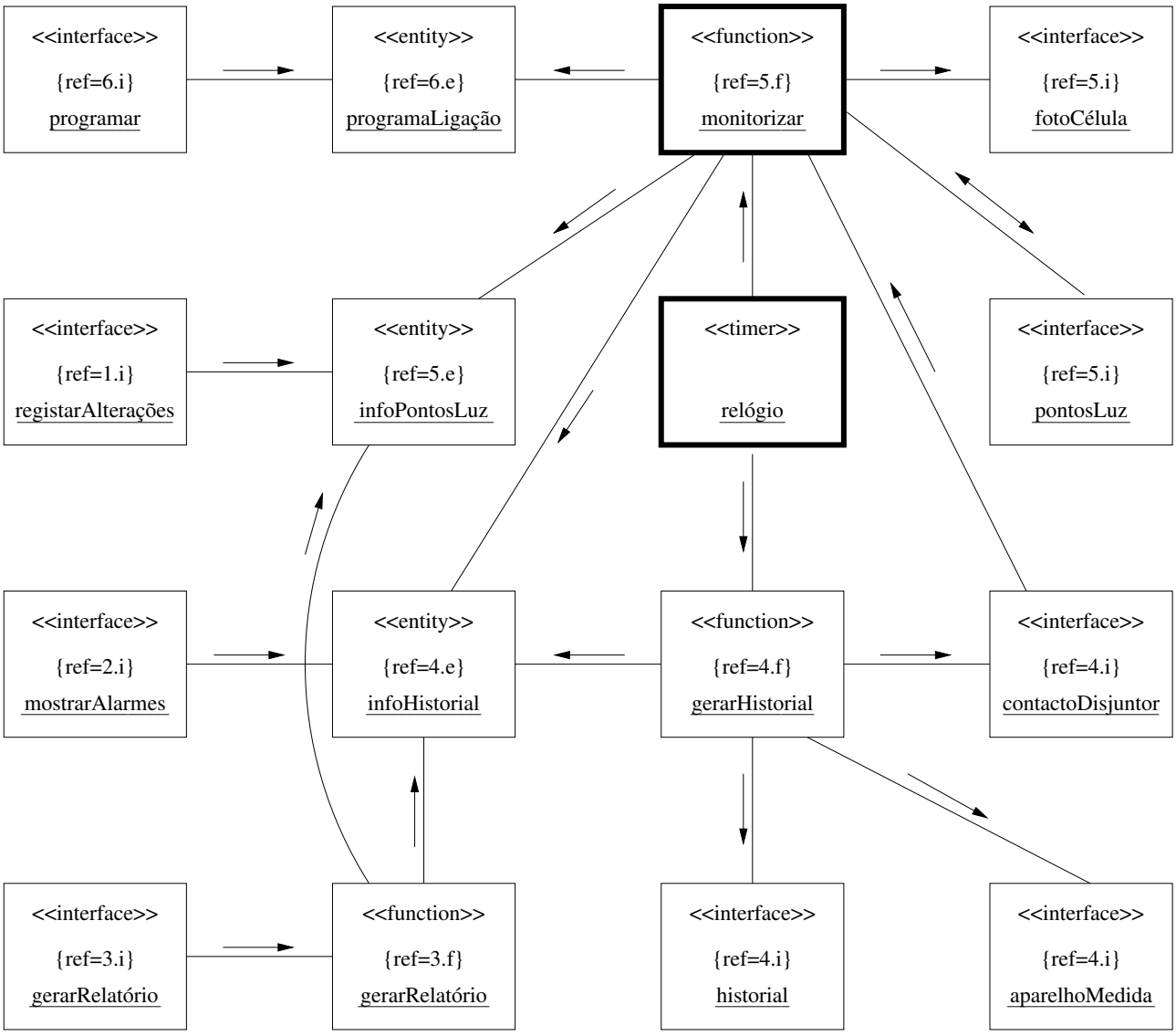


Figura 7.8: O diagrama de objectos do SSI.

Repare-se que, para os objectos, também se usam valores etiquetados {ref=n}. As referências usadas herdam o número do caso de uso a partir do qual foram criados e acrescentam um sufixo 'i', 'e' ou 'f', conforme se trate, respectivamente, dum objecto-interface, entidade ou função.

O SSI precisa dos dois tipos de informação temporal, indicados na pág. 140: eventos periódicos para *log* (pulso periódico gerado a cada hora) e informação do tempo absoluto para registar as ocorrências de falhas (hora e dia). Daí a inclusão dum objecto "relógio" que disponibiliza a informação temporal aos outros objectos. Este objecto tem aposto um estereótipo ≪timer≫ que é uma especialização do estereótipo ≪function≫.

Por vezes, é preciso modelar, internamente ao sistema, a informação ou o comportamento dum actor, mesmo sabendo que ele não faz parte do sistema. Por exemplo, num sistema de gestão de contas bancárias, são necessárias as propriedades relevantes dos clientes, apesar destes claramente não pertencerem ao referido sistema. Esta necessidade foi sentida relativamente aos pontos de luz, que apesar de terem sido correctamente identificados como actores, têm de ser modelados internamente para se poder fazer um inventário relativamente aos pontos de luz. Em resumo, esta estratégia para identificar objectos consiste em olhar para os actores do sistema e verificar se este manipula informação sobre aqueles. Se tal suceder, o sistema deve modelá-los como objectos internos.

Refira-se que a perspectiva que entende o desenvolvimento (leia-se, a construção dos diversos diagramas) como uma actividade iterativa, foi seguida durante o SSI. Por exemplo, a identificação do actor "contacto do disjuntor" só foi tida por necessária durante a construção do diagrama de objectos; anteriormente as informações provenientes desse actor e do aparelho de medida estavam agrupadas na mesma abstracção (num único fluxo), que provinha dum actor chamado aparelho de medida. Com a construção do diagrama de objectos, verificou-se que era vantajoso distinguir os dois actores, bem como os fluxos "valores eléctricos" e "estado do contacto".

O leitor atento verificará que não existe, no diagrama de objectos, o objecto {ref=1.e}, apesar dessa informação estar indicada no quadro 7.4. Sucede que, durante o processo de construção do diagrama de objectos, se chegou à conclusão que a informação manipulada (leia-se alterada) pelo caso de uso {ref=1} é a mesma que é necessária ao caso de uso {ref=5} e, portanto, por este criada. Assim sendo, considerou-se oportuno não estar a criar dois objectos-entidade, cuja informação armazenada e respectivo propósito fossem os mesmos.

Diagrama de classes e de state-charts

Para este exemplo, não foi tida por relevante a construção destes dois diagramas. Igualmente, a construção do repositório OBLOG não foi realizada, pois, como a seguir se indica, foi desenvolvido directamente em JAVA um protótipo do sistema, com o intuito de validar os requisitos do utilizador.

7.1.7 Concepção e implementação

Depois da fase de análise, seguem-se as fases de concepção e implementação que, neste projecto, foram realizadas ao mesmo tempo. Com base no diagrama de objectos obtido, foi elaborado, pelo autor, um protótipo em linguagem JAVA (ver código no apêndice A). A escolha desta linguagem, em detrimento de OBLOG, permite que se valide a utilização de outras linguagens como representações unificadas de sistemas embebidos. Como fontes de informação em relação à linguagem JAVA foram utilizados os seguintes livros: [Anuff, 1996] [Arnold e Gosling, 1996] [Naughton, 1996].

A estrutura de objectos foi fielmente seguida na construção do protótipo, o que demonstra

que os resultados obtidos durante a análise estão correctos, ou, se se preferir, adequados para serem usados nas fases subsequentes à análise. A utilização do protótipo (fig. 7.9), por parte do cliente, permitiu validar os requisitos deste, dando assim garantias que a versão final do programa (não realizada) serviria os interesses dos utilizadores finais do sistema.

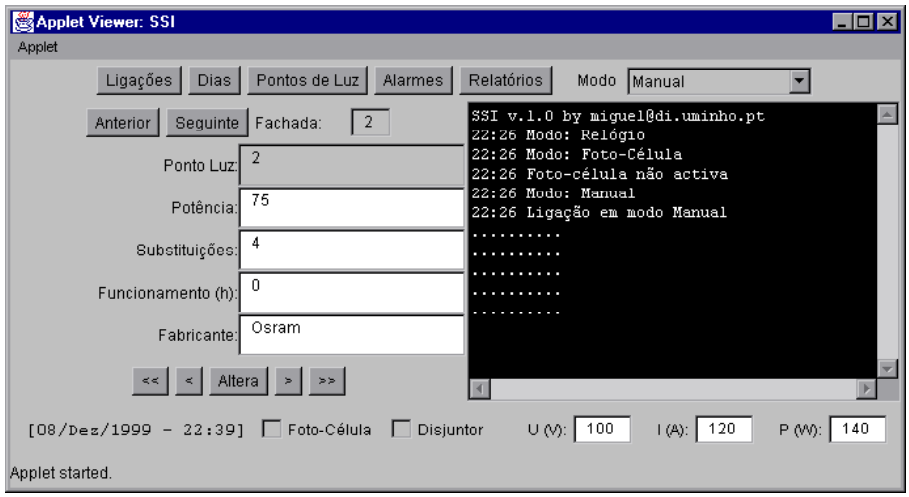


Figura 7.9: Aspecto do protótipo do SSI criado em JAVA.

Paralelamente à construção do protótipo em JAVA, foi construída uma aplicação que usou os resultados da fase de análise, como ponto de partida para uma implementação em C++ [Ferreira, 1999].

7.1.8 Comentários

Este exemplo real mostrou que o uso de objectos-interface é completamente crucial para poder alterar com facilidade a forma como os actores comunicam com o sistema. Revelou-se ainda muito útil para a realização de protótipos do sistema, pois permitiu que, numa primeira fase, todo o sistema funcionasse em software (simulando o funcionamento dos pontos de luz) e que depois os pontos de luz fossem substituídos por leds e interruptores. Se não fossem considerados os objectos-interface, seria necessário procurar por diversos objectos as questões de apresentação relacionados com um só objecto.

Para que a reutilização de objectos seja uma realidade e não uma mera hipótese conceptual, deve existir uma biblioteca de classes e um conhecimento profundo da sua constituição. Não foi mostrada essa possibilidade neste caso de estudo. Contudo, a diferença entre desenvolver um sistema da forma seguida nesta secção e numa situação onde houvesse o conhecimento duma biblioteca de classes reside apenas na seguinte facilidade: se depois de identificar um objecto, existir uma classe que o descreve, então esse objecto não necessita de ser implementado, podendo ser tirado da biblioteca de classes, eventualmente após ligeiras modificações [Sigfried, 1996, pág. 256].

7.2 Sistema de controlo das linhas HIDRO

A forma de apresentação deste caso prático será distinta daquela seguida na secção anterior, pois pretende mostrar-se o modo como o projecto foi realmente conduzido e tecer-se alguns comentários acerca das dificuldades e alterações que o processo de desenvolvimento sofreu relativamente às expectativas iniciais.

Refira-se que este projecto tem uma componente muito vincada de engenharia reversa, uma vez que o objectivo principal consistia em diagnosticar os problemas e as ineficiências dum sistema já em funcionamento e em optimizar esse sistema de acordo com os resultados desse diagnóstico.

7.2.1 O cliente

O Sistema de Controlo das Linhas HIDRO (SCLH) é responsável pela coordenação dum conjunto de linhas de produção de auto-rádios, instaladas na fábrica da BLAUPUNKT AUTO-RÁDIO PORTUGAL, LDA., situada na freguesia de Ferreiros, na cidade de Braga.

7.2.2 O problema

Por solicitação do cliente, que sentia que as suas linhas de produção apresentavam algumas ineficiências, foi criada uma equipa de projecto (designada por Team BP-UM e composta por elementos da BLAUPUNKT e do DI/UM), que tinha por missão fazer o diagnóstico e optimização do SCLH. Refira-se que a metodologia seguida no projecto foi, toda ela, determinada pelos elementos do DI/UM, incumbindo aos elementos da BLAUPUNKT validar os diversos documentos que foram sendo produzidos e fornecer informações necessárias à persecução do projecto.

7.2.3 Análise

Tratando-se este dum projecto com uma forte componente de engenharia reversa, a forma como ele foi faseado, ao longo do tempo, seguiu o modelo representado na fig. 7.10. Este modelo é meramente referencial, porque, por exemplo, as 1^a e 2^a fases foram, na realidade, realizadas parcialmente em simultâneo.

Nesta secção, descreve-se apenas o contributo que o autor deu às duas primeiras fases, por serem aquelas em que esteve mais directamente envolvido e, adicionalmente, por serem as que levantam questões mais relevantes no âmbito desta tese.

Os diagramas apresentados, nas próximas secções, são o resultado final de várias iterações que os mesmos sofreram ao longo do projecto [Team BP-UM, 1999a] [Team BP-UM, 1999b]. Nunca se considerou, no seio da equipa de desenvolvimento, que os documentos elaborados estivessem finalizados, pelo que alterações propostas em fases avançadas do projecto foram sempre aceites e obrigaram a actualizar todos os diagramas que delas dependiam.

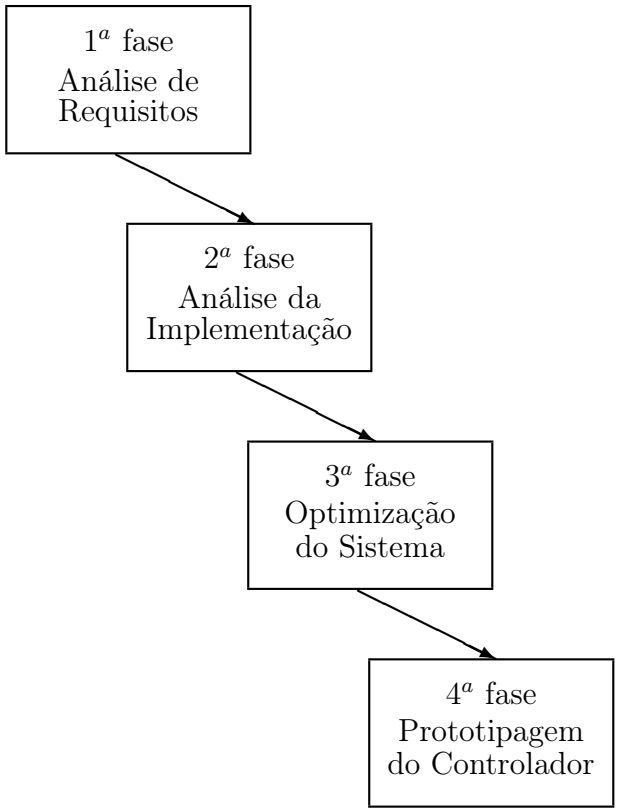


Figura 7.10: O faseamento do projecto.

7.2.4 Diagrama de contexto

Para criar um impacto visual maior, nomeadamente nos diagramas de contexto e de casos de uso, optou-se pela definição dum novo ícone para o actor auto-rádio, uma vez que se trata do objecto principal do negócio do cliente. Além disso, sendo o único actor não humano que foi identificado, ainda mais evidente resultou a necessidade de criar um novo ícone, de molde a rapidamente distinguir os actores humanos dos auto-rádios.

A fig. 7.11 mostra o diagrama de contexto do SCLH, onde se podem observar os actores que interagem com o sistema. Relativamente aos nomes atribuídos aos actores, houve o cuidado em evitar usar designações utilizadas pela Blaupunkt, pois compete a esta afectar a cada actor, um conjunto de colaboradores em concreto, de acordo com as políticas internas de gestão de recursos humanos.

Uma vez que alguns actores podem ser especializados, nomeadamente os operários, é também possível desenhar o diagrama de contexto que é apresentado na fig. 7.12, que deve ser visto como um refinamento do diagrama da fig. 7.12.

7.2.5 Diagramas de casos de uso

Depois de identificados os actores do SCLH, deve colectar-se a lista de casos de uso associados a esses actores:

1. manter/corrigir sistema.
2. configurar lugar.

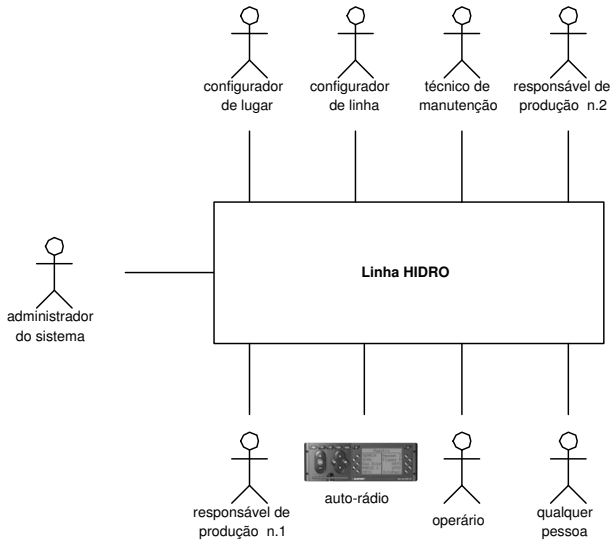


Figura 7.11: Diagrama de contexto para o SCLH.

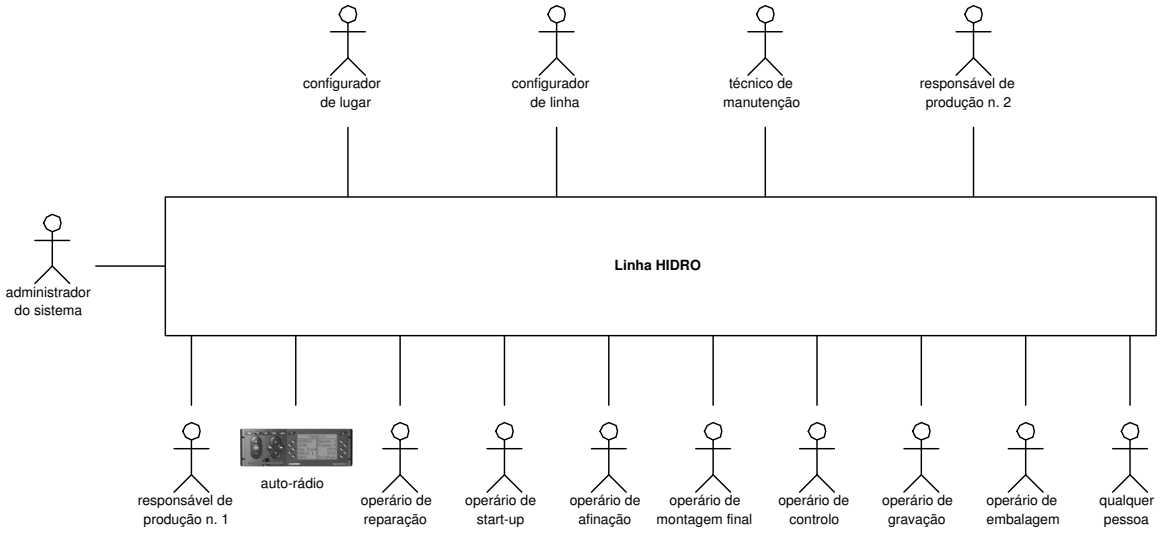


Figura 7.12: Diagrama de contexto mais refinado para o SCLH.

3. configurar linha.
4. activar equipamento.
5. recuperar de emergência.
6. gerar relatórios.
7. monitorizar linha.
8. gerir acessos ao sistema.
9. conduzir auto-rádio.
10. operar auto-rádio.
11. visualizar produção.
12. accionar emergência.

O diagrama de casos de uso de mais alto nível (i.e. de maior abstracção) é apresentado na fig. 7.13, em que se mostram quais os actores envolvidos em cada um dos casos de uso considerados. O quadro 7.5 lista as descrições textuais para cada um desses casos de uso.

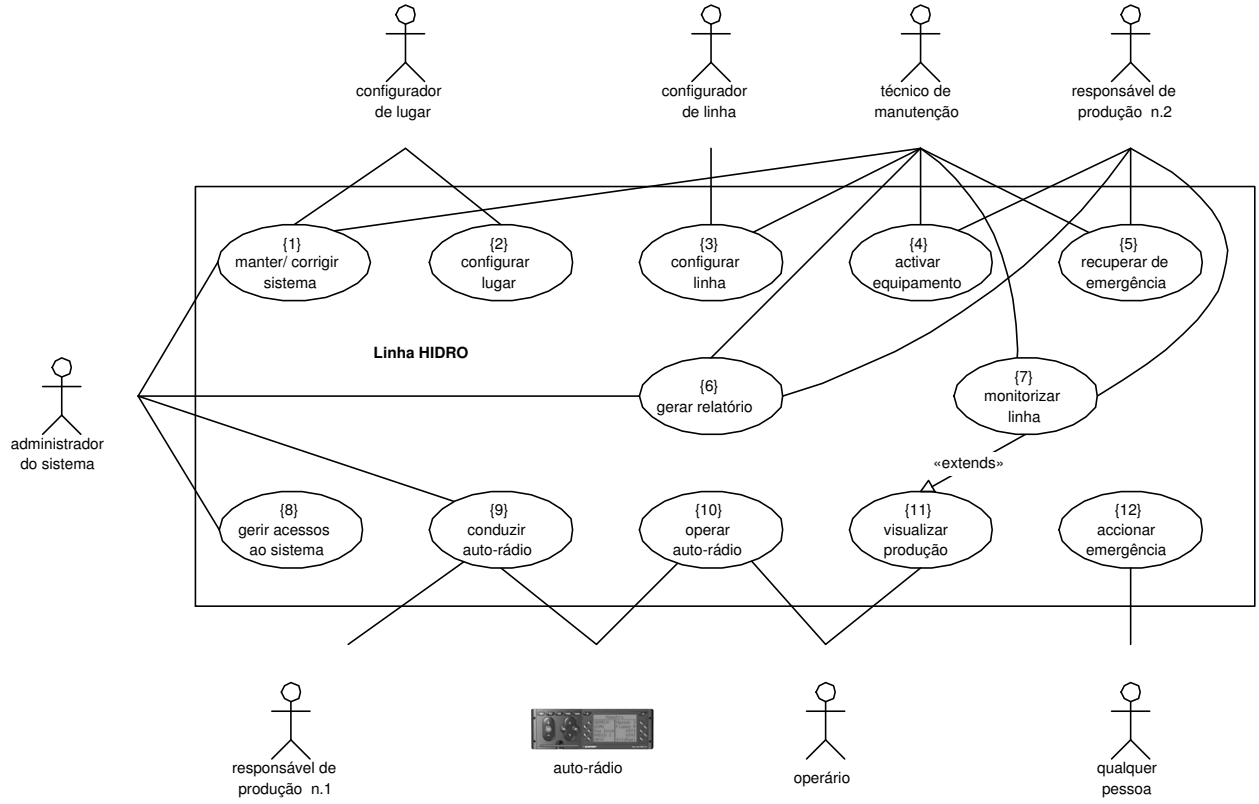


Figura 7.13: Diagrama de casos de uso para o SCLH.

Relativamente aos casos de uso 4, 9, e 10, achou-se conveniente em os especificar com mais pormenor. Nesse sentido, foram criados outros diagramas de casos de uso que, na prática, refinam cada caso de uso, dando origem a um novo diagrama de casos de uso contendo vários (sub-)casos de uso. Trata-se dum mecanismo bem conhecido que consiste em dividir uma dada funcionalidade em funcionalidades mais pequenas (decomposição funcional) e que permite lidar mais facilmente com a complexidade do problema em causa.

1. manter/corrigir sistema Permite manter operacional o sistema nos níveis expectáveis, através da substituição/reparação/alteração/configuração de componentes eléctricos, electrónicos, software e mecânicos.
2. configurar lugar Identifica univocamente os postos do lugar e disponibiliza as operações que devem ser executadas nos postos.
3. configurar linha Indica qual a sequência de operações (lugares) que cada tipo/série de auto-rádios deve percorrer durante a produção. Esta funcionalidade deve permitir que possam coexistir na linha diferentes tipos/séries de auto-rádios, podendo cada um ter uma sequência diferente da dos outros. Desta forma, será possível repor na linha auto-rádios semi-produzidos (ex. vindos duma outra fábrica) ou retirar da linha auto-rádios não totalmente produzidos (ex. para enviar para outra fábrica).
4. activar equipamento Liga/desliga os diversos componentes da linha (sistema de transporte, postos, etc.).
5. recuperar de emergência Retoma o funcionamento normal do sistema, após ter sido colocado num estado de alarme.
6. gerar relatórios Faculta dados sobre o sistema, a produção, os postos e os auto-rádios, relativos a um determinado intervalo de tempo.
7. monitorizar linha Faculta dados sobre o estado actual da linha.
8. gerir acessos ao sistema Atribui palavras de passe aos vários utilizadores do sistema, de modo a regular as funcionalidades a que cada um deles tem acesso.
9. conduzir auto-rádio Movimenta os auto-rádios ao longo da linha, de forma a disponibilizá-los aos postos para realizar as diferentes operações.
10. operar auto-rádio Realiza um conjunto de operações para produzir um auto-rádio.
11. visualizar produção Faculta dados e objectivos de produção de relevância para as operações executadas nos postos.
12. accionar emergência Actua um botão de alarme, posicionado num local visível e de fácil acesso, para parar todos os elementos eléctricos e pneumáticos da linha.

Tabela 7.5: Descrição sumária dos casos de uso de mais alto nível do SCLH.

As fig. 7.14 e 7.15 mostram os diagramas de casos de uso relativos aos casos de uso 4 e 9. Note-se que os actores de cada um desses diagramas são exactamente aqueles que interagem com o respectivo caso de uso na fig. 7.13.

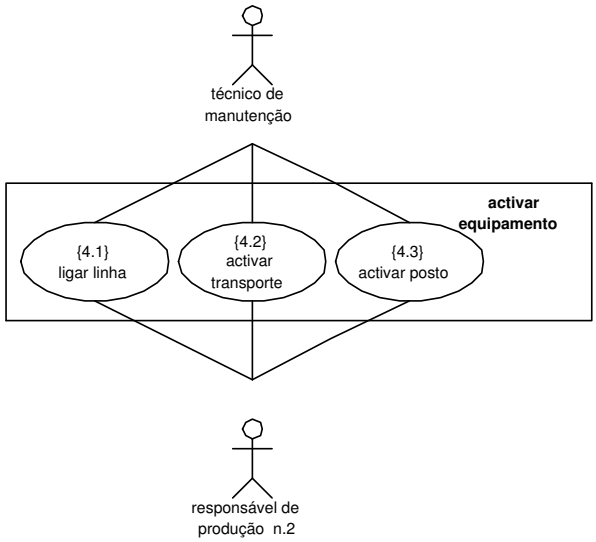


Figura 7.14: Diagrama de casos de uso para o caso de uso 4.

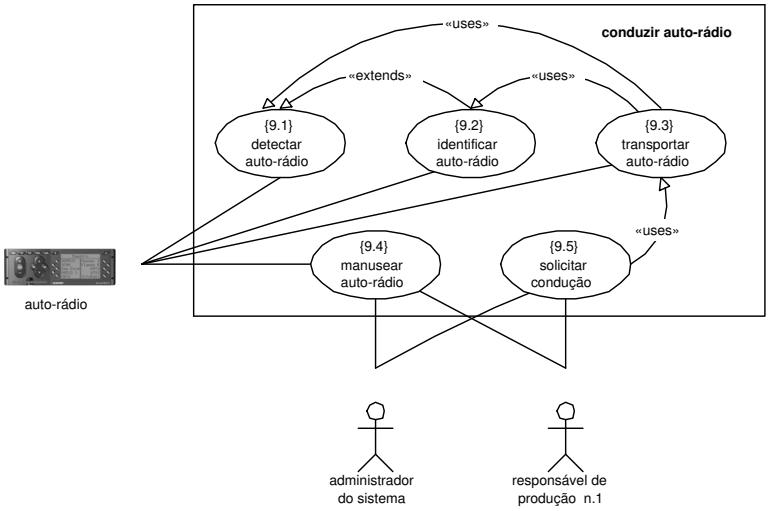


Figura 7.15: Diagrama de casos de uso para o caso de uso 9.

Em relação ao caso de uso 10, foram feitos dois refinamentos ortogonais. No primeiro (fig. 7.16), é feita uma divisão segundo a especialização, i.e. a funcionalidade que o caso de uso encerra é subdividida segundo as várias operações que são feitas ao longo da linha (montagem, afinação HFs, controlo, gravação, embalagem). O outro refinamento (fig. 7.17) é feito segundo uma decomposição da funcionalidade do caso de uso, i.e. o caso de uso é dividido funcionalmente em partes mais pequenas (menos abstractas).

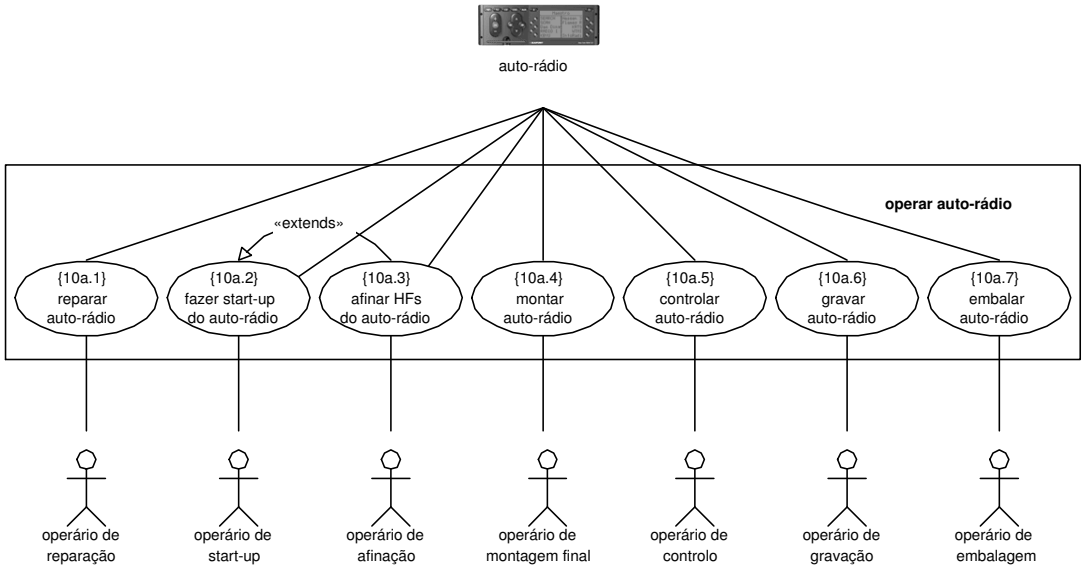


Figura 7.16: Diagrama de casos de uso para o caso de uso 10 (especialização).

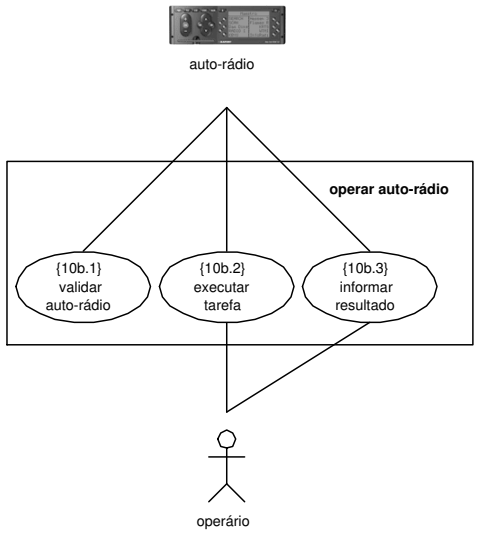


Figura 7.17: Diagrama de casos de uso para o caso de uso 10 (decomposição).

As descrições textuais, para os casos de uso apresentados nas fig. 7.14, 7.15 e 7.17, são listadas no quadro 7.6.

Seria também possível refinar, por exemplo, o caso de uso 10a.5, da forma que a fig. 7.18 documenta. No entanto, dado o grão demasiado fino que esse diagrama possui e a pouca influência que essa informação tem para o SCLH (não em termos de conteúdo, mas sim de pormenor), optou-se por não enveredar por este tipo de refinamento.

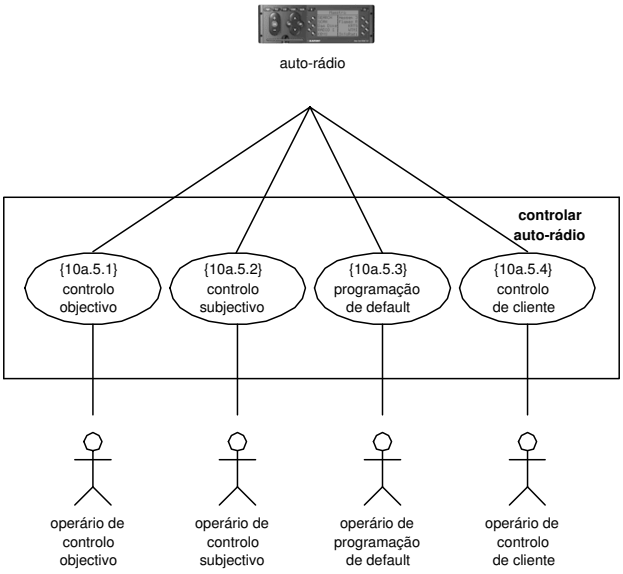


Figura 7.18: Diagrama de casos de uso para o caso de uso 10a.5.

Refira-se ainda que o SCLH deve ser tolerante a faltas relativamente ao incumprimento dos protocolos estabelecidos para acesso às funcionalidades da linha. Por outras palavras, além das funcionalidades descritas nos diagramas de casos de uso, o sistema deve ter também a aptidão para encaminhar correctamente um auto-rádio para o posto devido, independentemente do local da linha em que ele possa surgir (é admissível que ele apareça num local "impossível" por indevidamente alguém o ter transportado à mão). Igualmente, o sistema deve reagir e recuperar de situações em que alguém retira da linha um dado auto-rádio, enquanto este estava a ser encaminhado para um dado local (i.e. o sistema não deve bloquear, por ficar, indefinidamente, à espera que esse auto-rádio chegue ao local esperado).

Para cada uma das situações não desejáveis, mas previstas, deve ser guardada informação em histórico, de modo a ser possível, posteriormente, saber o que correu mal, quando e porquê.

7.2.6 Diagrama de objectos

Com base nos diagramas de casos de uso, constrói-se o diagrama de objectos. Neste exemplo, é necessário considerar o diagrama de casos de uso como a expansão do diagrama da fig. 7.13, com a informação que consta dos diagramas das fig. 7.14, 7.15 e 7.17. Por outras palavras, deve ver-se o diagrama de casos de uso como um diagrama plano (a um só nível) e não como estando organizado hierarquicamente.

Para cada caso de uso, fez-se um levantamento de quais as dimensões do espaço de análise

4.1. ligar linha Liga/desliga a alimentação eléctrica e pneumática da linha.
4.2. activar transporte Activa/desactiva os motores eléctricos do sistema de transporte da linha.
4.3. activar posto Autoriza/desautoriza o sistema de transporte a conduzir auto-rádios para o posto e/ou autoriza/desautoriza a execução das operações no posto. Por exemplo, num dado turno, um operário pode ter faltado, o que faz com que o respectivo posto não esteja ocupado, devendo, nessa situação, desactivar-se o posto.
9.1. detectar auto-rádio Determina a presença de um qualquer auto-rádio, num local preciso da linha, recorrendo a um conjunto de sensores de detecção (ex. ópticos, indutivos, capacitivos).
9.2. identificar auto-rádio Determina a presença de um determinado auto-rádio, num local preciso da linha, recorrendo a um conjunto de sensores de identificação (ex. leitores de código de barras).
9.3. transportar auto-rádio Encaminha, pelo sistema de transporte, o auto-rádio, com base no seu estado, na sua posição e na sequência de operações que lhe está associada. Devem existir dois modos de funcionamento distintos, um em que o auto-rádio é conduzido automaticamente e outro em que a condução é exterior à linha.
9.4. manusear auto-rádio Retira, com carácter temporário ou definitivo, ou (re)coloca um auto-rádio manualmente na linha. Quando se retira um auto-rádio da linha deve informar-se de imediato o sistema que tal operação foi realizada. Antes de se (re)colocar um auto-rádio na linha deve informar-se o sistema que tal operação irá ser realizada, indicando-se se o auto-rádio havia anteriormente sido retirado ou o seu estado, caso se trate dum auto-rádio novo. Esta funcionalidade evita que o sistema tenha de manipular um conjunto de informação muito grande, uma vez que há indicação explícita de quais os auto-rádios que são retirados e repostos na linha.
9.5. solicitar condução Força a condução compulsiva de um conjunto de auto-rádios para um dado posto de saída, a fim de serem retirados, temporária ou definitivamente, da linha. O conjunto de auto-rádios poderá ser designado por identificação (individual ou grupo) e/ou por estado. Esta funcionalidade deve ser usada quando, por exemplo, se detecta uma falha num tipo/série ou a indisponibilidade dum componente de montagem e se pretende cancelar ou suspender a sua produção. Tal como em 9.4., também é possível proceder à (re)colocação de auto-rádios na linha.
10b.1. validar auto-rádio Confirma se o auto-rádio que se encontra no posto pode ser operado.
10b.2. executar tarefa Executa sobre o auto-rádio que se encontra no posto a tarefa do posto.
10b.3. informar resultado Comunica ao sistema dados relevantes sobre a realização da tarefa.

Tabela 7.6: Descrição sumária dos casos de uso de segundo nível do SCLH.

que predominantemente cobriam. O quadro 7.7 indica, para os casos de uso considerados, quais as categorias de objectos que devem ser criadas. Refira-se novamente que os objectos-entidade, objectos-interface e objectos-função cobrem essencialmente as dimensões informação, apresentação e comportamento, respectivamente.

Caso de uso	interface	entidade	função
1. manter/corrigir sistema			
2. configurar lugar	X	X	
3. configurar linha	X	X	
4.1. ligar linha	X		
4.2. activar transporte	X		
4.3. activar posto	X	X	
5. recuperar de emergência	X		X
6. gerar relatórios	X	X	X
7. monitorizar linha	X		X
8. gerir acessos ao sistema	X	X	
9.1. detectar auto-rádio	X		
9.2. identificar auto-rádio	X		
9.3. transportar auto-rádio	X	X	X
9.4. manusear auto-rádio	X	X	
9.5. solicitar condução	X	X	
10b.1. validar auto-rádio	X		X
10b.2. executar tarefa	X	X	X
10b.3. informar resultado		X	
11. visualizar produção	X		X
12. accionar emergência	X		

Tabela 7.7: As categorias dos objectos para cada um dos casos de uso do SCLH.

Com a informação que consta do quadro 7.7 e das descrições textuais para os casos de uso, obteve-se o diagrama de objectos representado na fig. 7.19. Um diagrama de objectos pode conter pacotes ou subsistemas, sempre que seja necessário agrupar elementos de modelação (neste caso, objectos) em unidades conceptualmente mais abstractas [Booch et al., 1999, pág. 197]. Esta necessidade foi sentida, já que o diagrama inclui um número elevado de objectos, o que motivou que alguns desses objectos fossem agrupados em unidades lógicas (sub-sistemas).

Os estereótipos ≪sensor≫ e ≪actuator≫ são especializações do estereótipo ≪interface≫, e ≪black box≫ representa um objecto ou sub-sistema que faz parte do sistema, mas que, por algum motivo, não se pretende, no instante considerado, desenvolver. Este último estereótipo permite, em processos de reengenharia, marcar determinadas partes do sistema como inalteráveis.

Refira-se que o diagrama de objectos consubstancia uma visão mais global do que aquela que o SCLH foca. Esta *abordagem holística* de perspectivar os sistemas é extremamente importante, pois permite compreender o âmbito em que o SCLH está enquadrado. Desta forma, ao captar o contexto em que o sistema opera, podem ressaltar algumas questões que não seriam levantadas, caso o sistema fosse olhado segundo uma abordagem mais focalizada (mais restrita).

Se forem escondidos os pormenores internos dos vários pacotes que o diagrama de objectos

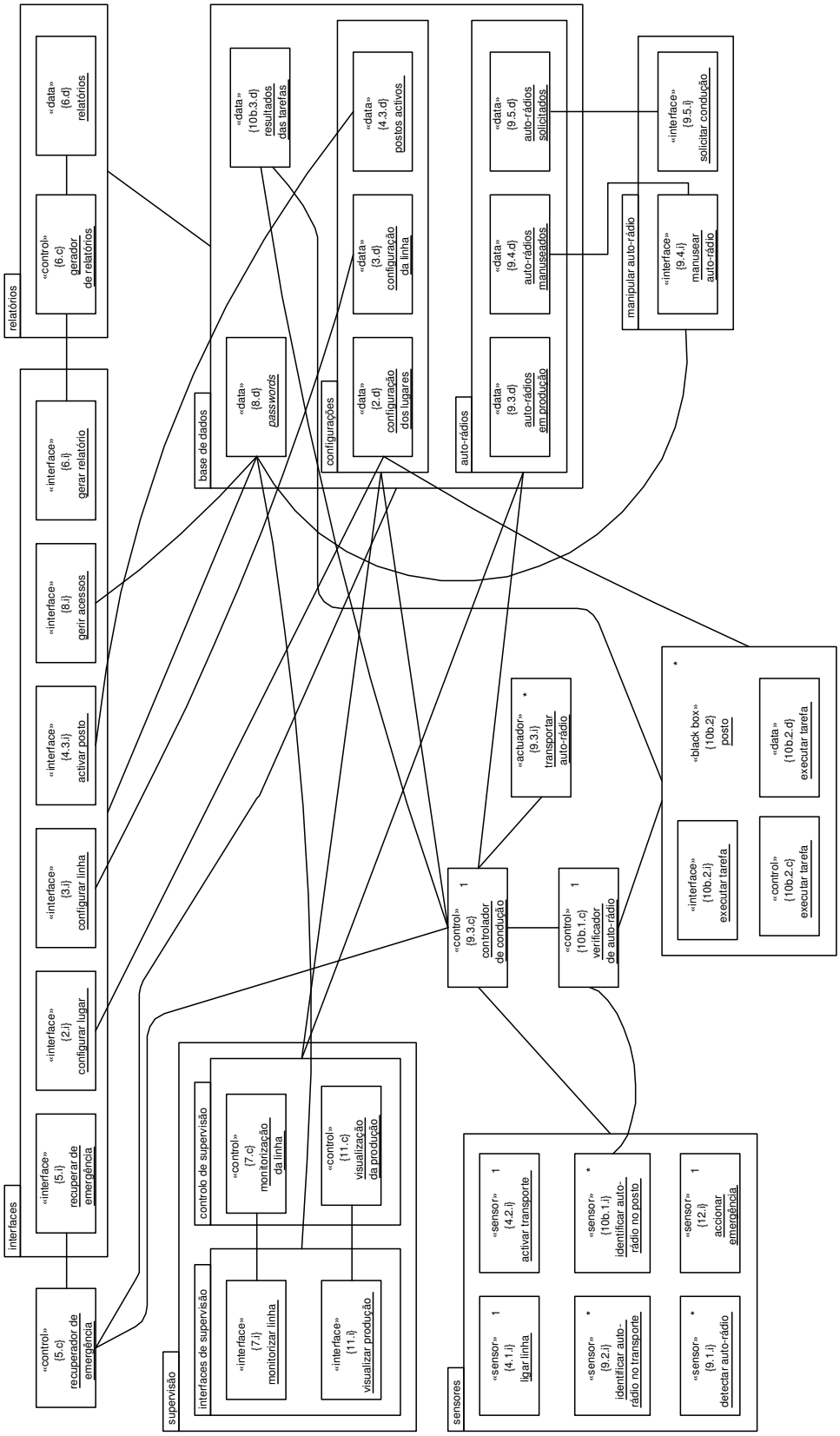


Figura 7.19: Diagrama de objectos para o SCLH.

inclui, pode obter-se o diagrama de objectos da fig. 7.20 que apresenta um nível de abstracção maior.

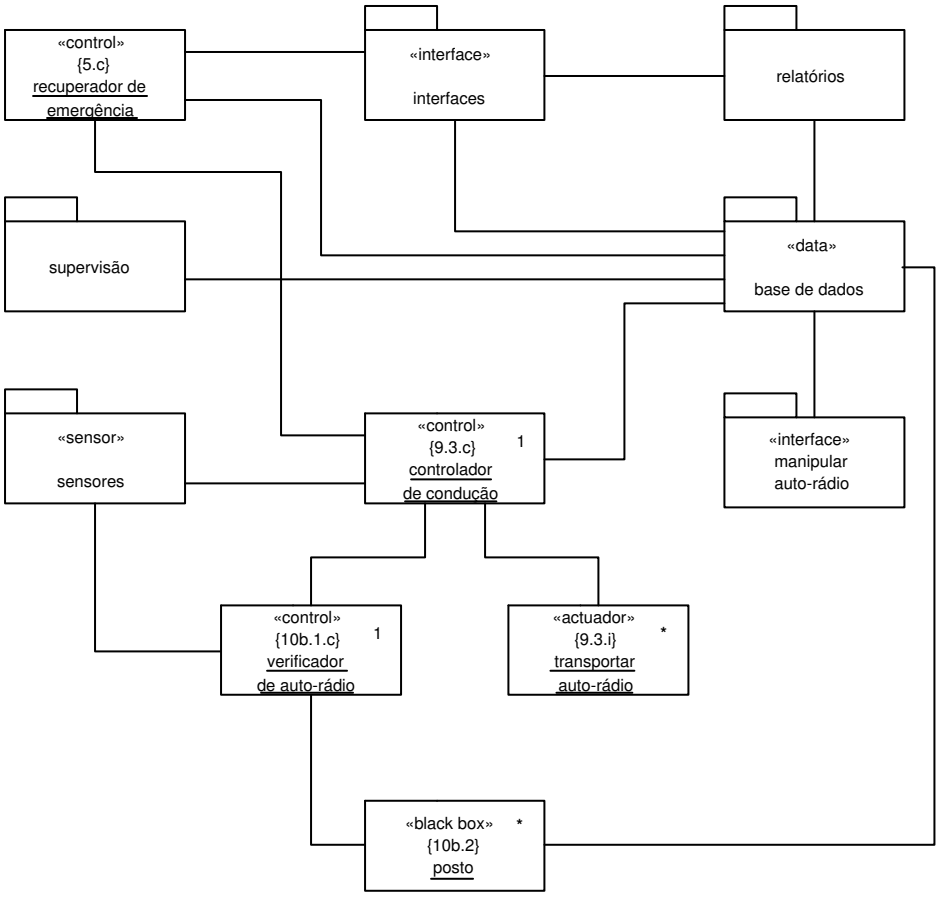


Figura 7.20: Diagrama (mais abstracto) de objectos para o SCLH.

7.2.7 Sistema controlado

As linhas HIDRO consistem em linhas de fabrico de auto-rádios em que o transporte e a respectiva decisão são executados automaticamente e sob a supervisão de um sistema de controlo (SCLH). A implementação actual do SCLH recorre a um autómato programável (PLC) e a um PC. O sistema de transporte das linhas HIDRO é composto por várias passadeiras rolantes e elevadores, por onde circulam paletes, em cima das quais se pousam os auto-rádios. É também possível que paletes vazias circulem ao longo da linha, nomeadamente quando um auto-rádio é embalado, pois a palete é reencaminhada para o início da linha para colocar um novo auto-rádio em produção.

As linhas HIDRO estão organizadas de modo a permitir um processamento sequencial (*pipeline*) dos auto-rádios, uma vez que as várias unidades de fabrico (postos) estão dispostas também em sequência (fig. 7.21). A única excepção a esta regra são os postos de reparação que estão situados na primeira posição da sequência, apesar de não fazerem parte do processamento sequencial principal, pois, idealmente, não é preciso reparar um auto-rádio. Os auto-rádios só são enviados para um posto de reparação, se for detectado algum problema ou avaria num dos

postos da sequência principal. Os blocos representados na figura (reparação, *start-up*, afinação HFs, montagem, controlo, gravação e embalagem) correspondem a zonas de processamento, podendo cada uma delas ser composta por vários postos de trabalho.

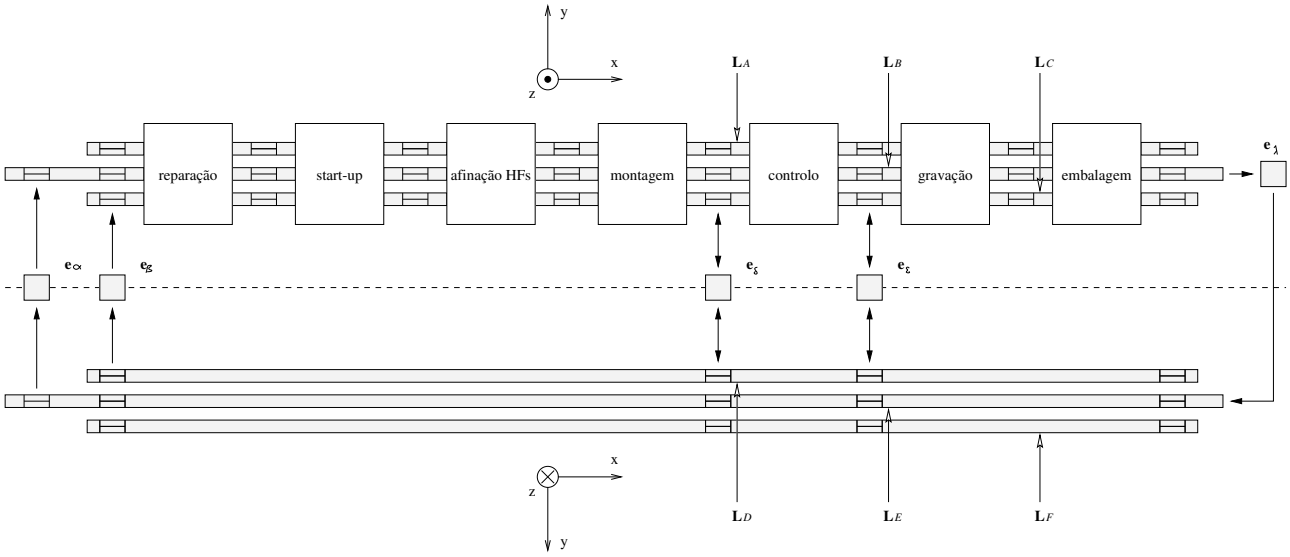


Figura 7.21: Esquema geral duma linha HIDRO.

As várias linhas HIDRO, instaladas na fábrica da BLAUPUNKT, não têm todas exactamente a mesma configuração, pelo que, no sentido de tentar ser o mais genérico possível, foi elaborada, no âmbito do projecto, uma notação que possibilita descrever, de forma fácil e rápida, as mais habituais alterações de linha para linha. Por exemplo, as diferenças mais flagrantes são o posicionamento dos elevadores e o número de linhas inferiores, embora outras menos evidentes também existam, como à frente se verá.

Cada linha HIDRO é composta por 5 ou 6 linhas de transporte de auto-rádios: 3 no plano superior (designadas por $\mathbf{L}_A$, $\mathbf{L}_B$, $\mathbf{L}_C$) e 2 ou 3 no plano inferior (designadas por $\mathbf{L}_D$, $\mathbf{L}_E$, $\mathbf{L}_F$). Nas linhas HIDRO em que só há 2 linhas inferiores, considera-se que não existe a linha $\mathbf{L}_F$. Cada linha é composta por um conjunto de passadeiras que tem um movimento uniforme, no sentido crescente do eixo Ox, para as linhas superiores e, no sentido decrescente do eixo Ox, para as linhas inferiores.

A linha superior $\mathbf{L}_B$ é usada essencialmente para transportar os auto-rádios entre postos não sequenciais e as linhas superiores $\mathbf{L}_A$ e $\mathbf{L}_C$ servem fundamentalmente para enviar os auto-rádios para os *buffers* dos postos (FIFOs, que se considera, que começam nos postos) e, quando não existe posto ou este está inactivo, têm a mesma função que a linha $\mathbf{L}_B$. A linha inferior $\mathbf{L}_E$ é utilizada para enviar paletes vazias até ao início da linha HIDRO, para serem subidas pelo elevador $\mathbf{e}_\alpha$, e as outras linhas inferiores $\mathbf{L}_D$ e, no caso de haverem 3 linhas inferiores, também $\mathbf{L}_F$ são usadas para transportar, para os postos apropriados através dos elevadores, os auto-rádios que não puderam anteriormente entrar no respectivo *buffer* por este estar cheio.

Os nós (compostos por transfers) permitem que os auto-rádios possam ser mudados de linha (no mesmo plano) ou que sejam enviados para os elevadores (para mudar de plano). Os elevadores estabelecem as ligações entre as linhas do plano superior e as linhas do plano inferior (na prática, os elevadores ligam directamente as linhas $\mathbf{L}_C$ e $\mathbf{L}_D$). Apesar dos elevadores

fazerem movimentos nos dois sentidos do eixo Oz, o transporte de paletes (com ou sem auto-rádios) pode ser restringido a apenas um sentido. Nas implementações actuais, existem cinco elevadores, designados de $\mathbf{e}_\alpha$, $\mathbf{e}_\beta$, $\mathbf{e}_\delta$, $\mathbf{e}_\varepsilon$, $\mathbf{e}_\lambda$. Os índices usados para os nós (a até i) e para os elevadores ($\alpha, \beta, \delta, \varepsilon, \lambda$) representam variáveis que há que concretizar com um dado valor (inteiro ou fraccionário).

O elevador $\mathbf{e}_\alpha$ transporta unicamente paletes vazias de $\mathbf{L}_E$ para $\mathbf{L}_B$, para que mais auto-rádios possam ser colocados, por intermédio do robô, nas linhas de transporte superior para serem processados.

O elevador $\mathbf{e}_\beta$ realiza transportes unicamente de $\mathbf{L}_D$ para $\mathbf{L}_C$, essencialmente para encaminhar auto-rádios avariados para os postos de reparação, ou para realimentar postos a jusante (todos os postos estão a jusante de $\mathbf{e}_\beta$) que, devido a *buffers* cheios, não aceitaram anteriormente mais auto-rádios.

Os elevadores $\mathbf{e}_\delta$ e $\mathbf{e}_\varepsilon$ realizam transportes nos dois sentidos:

1. de $\mathbf{L}_C$ para $\mathbf{L}_D$, essencialmente para encaminhar auto-rádios avariados para os postos de reparação, ou para realimentar postos a montante (reparação, start-up, afinação de HFs e montagem e, no caso de $\mathbf{e}_\varepsilon$, também controlo) que, devido a *buffers* cheios, não aceitaram mais auto-rádios;
2. de $\mathbf{L}_D$ para $\mathbf{L}_C$, essencialmente para realimentar postos a jusante (controlo, no caso de $\mathbf{e}_\delta$, e também gravação e embalagem) que, devido a *buffers* cheios, não aceitaram mais auto-rádios.

O elevador $\mathbf{e}_\lambda$ realiza transportes unicamente de $\mathbf{L}_B$ para $\mathbf{L}_E$, com o objectivo principal de encaminhar paletes vazias, libertadas por auto-rádios entretanto embalados, até ao início da linha HIDRO e, eventualmente, para realimentar postos a montante (gravação e embalagem) que, devido a *buffers* cheios, não aceitaram mais auto-rádios, ou para encaminhar auto-rádios avariados para os postos de reparação.

A colocação dos auto-rádios, provenientes das fases de fabrico anteriores, na linha $\mathbf{L}_B$ junto ao elevador $\mathbf{e}_\alpha$, é executada por um robô.

A presença duma palete é detectada por sensores, e um auto-rádio é detectado e identificado por, respectivamente, sensores ópticos e por leitores de código de barras.

A apresentação do equipamento de fabrico das linhas HIDRO, feita nesta secção, foi muito superficial. No apêndice B pode encontrar-se informação mais completa e elaborada sobre os diversos elementos (sensores e actuadores) que constituem o sistema controlado das linhas HIDRO.

7.2.8 Estratégias de controlo

A especificação do controlo ideal para condução de paletes (com ou sem auto-rádios), ao longo das linhas HIDRO, reclama a caracterização e a definição prévias das estratégias que devem ser seguidas. Neste sentido, foram identificados 2 níveis de controlo relativamente à movimentação das paletes.

O controlo de nível 1 é responsável pelas decisões estratégicas de transporte das paletes, enquanto que no controlo de nível 2 são tomadas decisões que permitem a movimentação em concreto das paletes, ao longo das linhas HIDRO.

Esta divisão pressupõe que o objecto 9.3.c seja dividido, dando origem a dois controladores (um

para cada nível de controlo). O controlador de nível 2 pode, por sua vez, ser dividido em tantos sub-controladores, quanto o número de nós compostos superiores e inferiores. Quando um nó composto superior e um nó composto inferior estão ligados por um elevador, considera-se, para efeitos de controlo, esse conjunto como um único sub-controlador.

Embora ambos os níveis de controlo possuam as suas regras próprias que caracterizam o seu comportamento específico, qualquer um desses níveis tem por objectivo fundamental e genérico contribuir para o cumprimento das seguintes meta-regras:

mr-1 As linhas de transporte devem transportar as paletes para o seu destino, sem enganos e no menor tempo possível, garantindo uma taxa (quantidade de auto-rádios por unidade de tempo) pré-definida de produção de auto-rádios.

mr-2 As linhas de transporte devem funcionar concorrentemente, desde que o paralelismo não ponha em causa o cumprimento dos objectivos de eficiência.

mr-3 Os transfers e os elevadores são considerados recursos escassos do sistema controlado, devendo ser geridos como tal, ou seja, deve ser minimizado o seu tempo de afectação às paletes e devem ser eficazmente controlados os pedidos de acesso múltiplo.

mr-4 Nos transfers intra-zona (da mesma zona), deve minimizar-se o tempo de decisão, uma vez que uma zona é composta por postos que realizam a mesma tarefa.

mr-5 Nos transfers inter-zona (de zonas diferentes), deve obedecer-se a um compromisso entre a maximização da exactidão de decisão e a minimização do tempo de decisão.

mr-6 Deve existir a garantia de cumprimento da sequência de processamento de cada auto-rádio.

mr-7 O sistema de controlo deve poder determinar o mais exactamente possível em que local das linhas se encontra cada um dos auto-rádios em processamento.

mr-8 Deve existir um histórico com os percursos e mudanças de estados de todos os auto-rádios em produção, bem como com todas as situações anómalas detectadas pelo sistema de controlo.

mr-9 O sistema de informação das linhas HIDRO deve calcular, aquando da configuração das linhas, o número de paletes a colocar em trânsito (deverá igualmente existir um reajustamento dinâmico desse valor de acordo com o estado da linha em cada momento).

mr-10 Não deve ser permitida a circulação nas linhas de auto-rádios com códigos iguais.

mr-11 Se o controlo de nível 1 falhar (não existindo a informação das zonas/postos para onde cada auto-rádio deve ser encaminhado), as linhas HIDRO devem ser controladas unicamente pelo nível 2 (circulando as paletes pelas linhas do meio), não existindo, eventualmente, encravamento nos postos pela impossibilidade de execução do caso de uso 10b.1.c.

mr-12 Se o controlo de nível 2 falhar (não existindo forma de actuar sobre os objectos do sistema controlado para encaminhar as paletes para os locais pretendidos), as linhas HIDRO devem ser controladas unicamente pelo nível 1 (mantendo-se o encravamento nos postos através da execução do caso de uso 10b.1.c), realizando-se o transporte das paletes, por exemplo, com carrinhos.

Controlo de Nível 1

Neste nível são tomadas, essencialmente, as decisões de condução dos auto-rádios (paletes com auto-rádios) que garantem que cada auto-rádio percorre as zonas que fazem parte da sua sequência de processamento. Este nível define quatro conjuntos de regras (cálculo da próxima zona, processamento nos postos, tratamento de *buffers* e encaminhamento estratégico).

As **regras para cálculo da próxima zona** pretendem definir de que forma e em que circunstâncias o controlo de nível 1 deve efectuar aquele cálculo:

rpz-1 O controlo de nível 1 deve receber dos postos o resultado de processamento de cada auto-rádio, imediatamente após o auto-rádio ter sido processado no posto.

rpz-2 Para cada auto-rádio, após ter sido processado por uma zona e antes de entrar fisicamente noutra (próximo nó de decisão), o controlo de nível 1 deve calcular qual a próxima zona de processamento.

rpz-3 Para cada auto-rádio, o cálculo da próxima zona de processamento exige o conhecimento da sua identidade, do seu tipo, do resultado do processamento efectuado na zona anterior e da configuração da linha (previamente efectuada).

As **regras de processamento nos postos** pretendem definir em que condições os auto-rádios devem ser autorizados, pelo controlo de nível 1, a ser processados nos postos:

rpp-1 O controlo de nível 1, sob pedido dos postos, deve autorizar previamente cada auto-rádio a ser processado nos postos.

rpp-2 Um auto-rádio pode ser processado num posto, se este pertencer à sequência de processamento, se a tarefa do posto ainda não foi executada com sucesso sobre o auto-rádio, se existe a garantia de que sobre o auto-rádio foram realizadas com sucesso as tarefas anteriores e se o auto-rádio chegou fisicamente ao posto em causa sob coordenação do sistema de controlo das linhas.

As **regras de tratamento dos buffers** pretendem definir a forma como o controlo de nível 1 deve gerir os *buffers* dos vários postos existentes nas linhas Hidro:

rtb-1 Numa mesma zona, os *buffers* dos diversos postos devem ser preenchidos com auto-rádios, de acordo com as prioridades que estão associadas aos postos (por configuração inicial).

rtb-2 Qualquer posto que não processe auto-rádios (com ou sem sucesso), durante um determinado tempo (configurável inicialmente), deve ser automaticamente inactivado e o seu *buffer* esvaziado, desde que não seja o único posto activado da zona a que pertence.

rtb-3 Quando um *buffer* dum posto não receber auto-rádios, durante um determinado tempo (configurável inicialmente), o posto deve ser automaticamente inactivado, desde que não seja o único posto activado da zona a que pertence.

As **regras de encaminhamento estratégico** pretendem definir a forma como o controlo de nível 1 deve encaminhar os auto-rádios, ao longo das linhas de transporte, garantindo o cumprimento das sequências de processamento de cada um dos auto-rádios em produção:

ree-1 Se um auto-rádio sair de um posto sem que o resultado do seu processamento tenha ainda chegado ao controlo de nível 1, o auto-rádio deve ser encaminhado para um dado posto de saída. Enquanto for fisicamente possível reencaminhar o auto-rádio para a próxima zona de processamento, o sistema de controlo de nível 1 deve processar o resultado do processamento do auto-rádio (caso o resultado chegue entretanto), anulando o envio do auto-rádio para o posto de saída. O histórico deve ser informado de todos os estados pós-processamento do auto-rádio em causa.

ree-2 Se o último resultado de processamento de um auto-rádio, num qualquer posto, for sem sucesso, o auto-rádio, após ser reposto nas linhas de transporte, deve ser encaminhado para a zona de reparação.

ree-3 Um auto-rádio, após ter sido reparado e ter sido reposto nas linhas de transporte, deve ser encaminhado para a zona de processamento que originou o seu envio para reparação.

ree-4 Se um auto-rádio, após ter sido processado com sucesso na sua última zona (tipicamente na embalagem), aparecer fisicamente na linha, deve ser encaminhado para um dado posto de saída.

ree-5 Porções de linha de transporte que não contenham no seu interior postos activos deverão, se necessário, ser utilizadas como percursos alternativos de transporte (*vias rápidas*).

ree-6 Sob pedido do controlo de nível 2 e mediante indicação do valor do leitor do código de barras, o controlo de nível 1 deve fornecer o destino de condução, dentro de cada nó, de uma palete.

ree-7 Sob pedido do controlo de nível 2, o controlo de nível 1 deve autorizar uma palete a entrar para um transfer, se o trajecto que deve executar não se cruza com nenhuma palete já em trânsito no nó e se à saída do último transfer do seu percurso existe espaço para ela.

As regras deste nível são responsáveis por determinar a zona destino dum auto-rádio, assim que uma dada operação, para esse auto-rádio, for dada como terminada (independentemente do sucesso ou insucesso da operação realizada).

No fim de cada tarefa realizada a um auto-rádio num posto, e com base no respectivo resultado, deve determinar-se qual o destino desse auto-rádio. O destino dum auto-rádio é apenas a zona para a qual ele deve ser a seguir enviado e depende da configuração de linha para a série a que esse auto-rádio pertence.

Para qualquer posto, com a excepção dos que pertencem à zona de embalagem, o algoritmo a aplicar é o seguinte:

```
SE sucesso ENTÃO
  destino1 = Succ(ZonaPosto)
SENÃO
  destino1 = Reparação
```

A função `Succ` devolve a zona seguinte tendo em conta a zona a que pertence o posto actual e a configuração da linha para a série a que pertence o auto-rádio. Considerando a fig. 7.21 e admitindo um auto-rádio que pertence a uma série cuja configuração obriga a percorrer todas as zonas, quando a operação realizada num posto de Afinação HFs terminar com sucesso, a função `Succ` deve devolver como resultado `Montagem`, pois é essa a zona seguinte para a qual o auto-rádio deve ser encaminhado.

Como excepção ao algoritmo anterior, tem-se um posto de embalagem, por ser o último da sequência de fabrico. Para esse posto, deve usar-se o seguinte algoritmo:

```
SE sucesso ENTÃO
  apagar auto-rádio da base dados
SENÃO
  destino1 = Reparação
```

Este nível de controlo, além de determinar, a pedido e num dado momento, a zona de destino dum dado auto-rádio, é também responsável por regular o acesso dos auto-rádios aos recursos do sistema controlado. Assim sendo, o controlo de nível 1 deve definir qual a estratégia de prioridade, quando há, no mesmo instante (leia-se, mesmo impulso do sinal de sincronismo) e para o mesmo recurso do sistema controlado (por exemplo, um transfer), mais do que um auto-rádio a requerer a sua utilização.

Controlo de Nível 2

Este nível necessita de informação de controlo fornecida pelo nível 1, sendo, essencialmente, responsável pelo controlo dos percursos das paletes ao longo do sistema controlado, ou seja, executa de facto as decisões de condução tomadas pelo nível 1.

Desta forma, neste nível de controlo estão definidos quatro conjuntos de regras (acesso aos transfers, acesso aos elevadores, tolerância a faltas e optimização do desempenho) que devem ser fielmente implementadas pelo controlador das linhas.

As **regras para acesso aos transfers** pretendem definir a estratégia global das movimentações físicas de paletes ao longo dos transfers:

rat-1 Quando o controlo de nível 2 detectar uma palete à entrada de um transfer deve ler o valor do leitor de códigos de barras e, fornecendo-o ao controlo de nível 1, deve solicitar a este o destino de condução da palete e a autorização de entrada da palete no transfer.

rat-2 O controlo de nível 2 deve fazer entrar a palete para um transfer, após o controlo de nível 1 ter fornecido o destino de condução da palete e ter dado autorização para executar a trajectória.

rat-3 A movimentação das paletes através dos transfers deve ser concorrente, desde que os percursos exigidos por cada uma das paletes não sejam cruzados (coincidentes, total ou parcialmente, em relação aos transfers), mesmo que com algum desfasamento temporal.

rat-4 A movimentação concorrente de paletes, através dos transfers, deve possuir pontos de sincronismo, em que se deve garantir que não existem objectos colocados nos transfers ou elevadores; ou seja, não se deve permitir iniciar movimentações concorrentes em transfers de um mesmo nó, se existirem já um ou mais movimentos bloqueados num dos transfers.

As **regras para acesso aos elevadores** pretendem definir a estratégia global das movimentações físicas de paletes, ao longo dos elevadores:

rae-1 Quando o controlo de nível 2 receber, do controlo de nível 1, como destino de condução de uma palete, o índice de uma linha pertencente a um plano (superior ou inferior) diferente do da linha em que se encontra a palete a conduzir, torna-se necessário encaminhar a palete por intermédio de um elevador.

rae-2 O encaminhamento de uma palete que exija a utilização de um elevador deve ser realizado em duas fases distintas; a primeira em que se assume como destino final o transfer que se encontra dentro do elevador (devolvendo o controlo de nível 1, aquando do pedido inicial do controlo de nível 2, uma autorização de início de encaminhamento, no âmbito do cumprimento de ree-7, assim que este semi-percurso se encontrar livre); a segunda em que se assume como destino final o verdadeiro destino da palete, iniciando-se no transfer que se encontra dentro do elevador (devendo o controlo de nível 1 ser solicitado pelo controlo de nível 2 para autorizar o início do encaminhamento, no âmbito do cumprimento de ree-7, para cumprir o segundo semi-percurso).

As **regras para tolerância a faltas** pretendem definir procedimentos importantes para dotar o sistema de controlo das precauções adequadas para evitar o aparecimento de situações não desejadas (*deadlock*, *starvation*, *non-reachability*).

A *tolerância a faltas* revela-se uma característica extremamente pertinente neste sistema, pois, apesar de estarem definidos os protocolos de acesso às funcionalidades da linha, não é realista, nem prudente, assumir que os mesmos não serão violados (quer propositadamente, quer por pressões de diversa ordem).

rtf-1 Todas as leituras de sensores devem ter a possibilidade de evitar *glitches* com durações configuráveis.

rtf-2 Todas as leituras de sensores devem ter a possibilidade de não serem bloqueantes para o fio de controlo que a processa, através da definição de temporizadores de leitura.

rtf-3 O controlo de nível 2 não deve bloquear à espera das respostas do controlo de nível 1, decidindo encaminhar as paletes à entrada de transfers para as linhas de transporte do meio (linha B, para os nós compostos superiores, e linha E, para os nós compostos inferiores) ou para vias rápidas alternativas, se ao fim um determinado tempo não receber resposta do controlo de nível 1.

rtf-4 A leitura dos códigos, utilizando os leitores de códigos de barras, deve ser desencadeada por um sinal (de *trigger*) enviado pelo controlador, para evitar perdas de sincronismo na recepção dos códigos lidos.

rtf-5 Para todos os movimentos físicos de paletes, através de transfers ou de elevadores, mesmo que previamente autorizados pelo controlo de nível 1, o controlo de nível 2 deve verificar: (i) a existência de palete para movimentar (a palete pode ter sido retirada da linha à mão); (ii) a inexistência de objectos no local para onde se pretende movimentar a palete (pode ter sido colocado à mão algum objecto no local para onde se pretende movimentar a palete); (iii) a chegada da palete ao local para onde se desencadeou o movimento (a palete pode ter sido retirada da linha à mão, durante a movimentação).

rtf-6 Sempre que existir uma movimentação física de uma palete, deve ser gerada uma mensagem (notificando o facto) e enviada à base de dados (que a armazenará nos históricos do sistema).

rtf-7 Para todas as situações irregulares detectadas (colocação de objectos na linha à mão, ou retirada de paletes da linha à mão) deve ser gerada uma mensagem (referindo o facto) e enviada à base de dados (que a armazenará nos históricos do sistema).

As **regras para optimização do desempenho** pretendem definir decisões que devem ser tomadas para evitar a degradação do desempenho do sistema (ou seja, para melhorar o desempenho do sistema ideal aquando do surgimento de situações que obrigariam o sistema a percorrer estados de controlo menos eficazes), mesmo que, algumas vezes, à custa do incumprimento directo ou imediato de algumas regras explícitas de encaminhamento de paletes:

rod-1 Sempre que um leitor de código de barras devolver um código de identificação de auto-rádio inválido (podendo significar inexistência de auto-rádio ou etiqueta com código de barras mal posicionada ou mal identificada), e desde que não exista mais nenhum sensor para caracterizar melhor a situação, nem se esteja num nó com acesso a uma via rápida para encaminhamento à zona de reparação, a palete deve ser encaminhada para as linhas de transporte do meio (linha B, para os nós compostos superiores, e linha E, para os nós compostos inferiores) ou para vias rápidas alternativas.

rod-2 Quando uma palete não puder sair de um transfer devido ao facto do seu próximo local de movimentação (um transfer, um elevador, ou uma linha de transporte) estar ocupado, devem gerar-se ciclos consecutivos de temporizações, durante os quais a palete deve esperar que o local de movimentação seja desocupado. No final de cada ciclo de temporização, se o local de movimentação continuar ocupado e surgir uma segunda palete imediatamente a montante do transfer onde se encontra a primeira palete ou se o local de movimentação continuar ocupado e, não surgindo uma segunda palete, existir mais um posto activo da mesma zona, a primeira palete deve ser encaminhada para as linhas de transporte do meio (linha B, para os nós compostos superiores, e linha E, para os nós compostos inferiores) ou para vias rápidas alternativas.

rod-3 Sempre que uma palete, ao longo do seu percurso pelos transfers, estiver num transfer de uma linha de transporte do meio (linha B, para os nós compostos superiores, e linha E, para os nós compostos inferiores) ou de uma via rápida alternativa, deve ser verificado, durante um tempo pré-definido, se o local à saída do último transfer do seu percurso se encontra ocupado; caso esse local esteja ocupado a palete deve ser descarregada para a linha de transporte.

As regras de nível 2 permitem que se determine para que linha deve ser enviado um auto-rádio, quando detectado a montante dum nó composto, com base na zona destino calculada no nível 1 e no estado dos *buffers* dos postos activos (o esquerdo e o direito) desse nó.

A este nível, os nós são divididos em superiores e inferiores, uma vez que a decisão de qual o destino dum auto-rádio depende apenas, a este nível, do plano (superior ou inferior) do nó em causa e da existência dum elevador que permita enviar paletes para o outro plano. Embora se pudesse ter considerado nós completos (com linhas superiores e inferiores), a divisão assumida permite obter regras mais simples (embora em maior número).

A formalização das regras anteriormente apresentadas é sintetizada num conjunto de diagramas de state-charts (subsecção 7.2.12), que descrevem o comportamento (algoritmo) pretendido para os controladores de nível 2.

Simulações

No âmbito deste projecto, foi desenvolvida, em colaboração com o Dep. Produção e Sistemas da U.Minho, uma aplicação de simulação com o principal objectivo de validar as estratégias de controlo (de ambos os níveis) propostas nesta secção, em função sobretudo do número de auto-rádios produzidos por unidade temporal. A fig. 7.22 mostra o aspecto da interface com o utilizador da aplicação desenvolvida.

A referida aplicação, desenvolvida no ambiente Arena [Kelton et al., 1998], modela actualmente um nó composto superior com 3 linhas e, com base numa série de valores estatísticos totalmente parametrizáveis (taxa de chegada de auto-rádios, tempo de processamento nos postos, tempos de movimentação dos auto-rádios, etc.) tidos por relevantes para caracterizar a linha de produção, simula o seu comportamento. Como resultado são gerados alguns valores (ocupação dos postos, tamanho dos *buffers*, tempo médio de espera, etc) que permitem aferir o impacto das estratégias de controlo no funcionamento do sistema.

A simulação pode ser executada com as estratégias de controlo da implementação actual, obtidas na 2^a fase do projecto (fig. 7.10), ou com as novas estratégias definidas nesta secção. Desta forma, é possível comparar a produção de auto-rádios nos dois cenários (actual e futuro), o que permite à equipa de desenvolvimento escolher uma estratégia que garanta uma taxa de produção de auto-rádios superior (ou pelo menos igual) à actual. Este dado era uma imposição do cliente que pretendia um sistema mais robusto, mais fiável, mais seguro, com maior tolerância a faltas, mais flexível, mas simultaneamente com uma capacidade de produção, na pior das hipóteses, igual à actual.

7.2.9 Diagrama de objectos revisitado

Com base no diagrama de objectos da fig. 7.19, é possível obter o diagrama de objectos final (fig. 7.23). Este diagrama reflecte os limites do controlador do sistema, tendo em atenção a

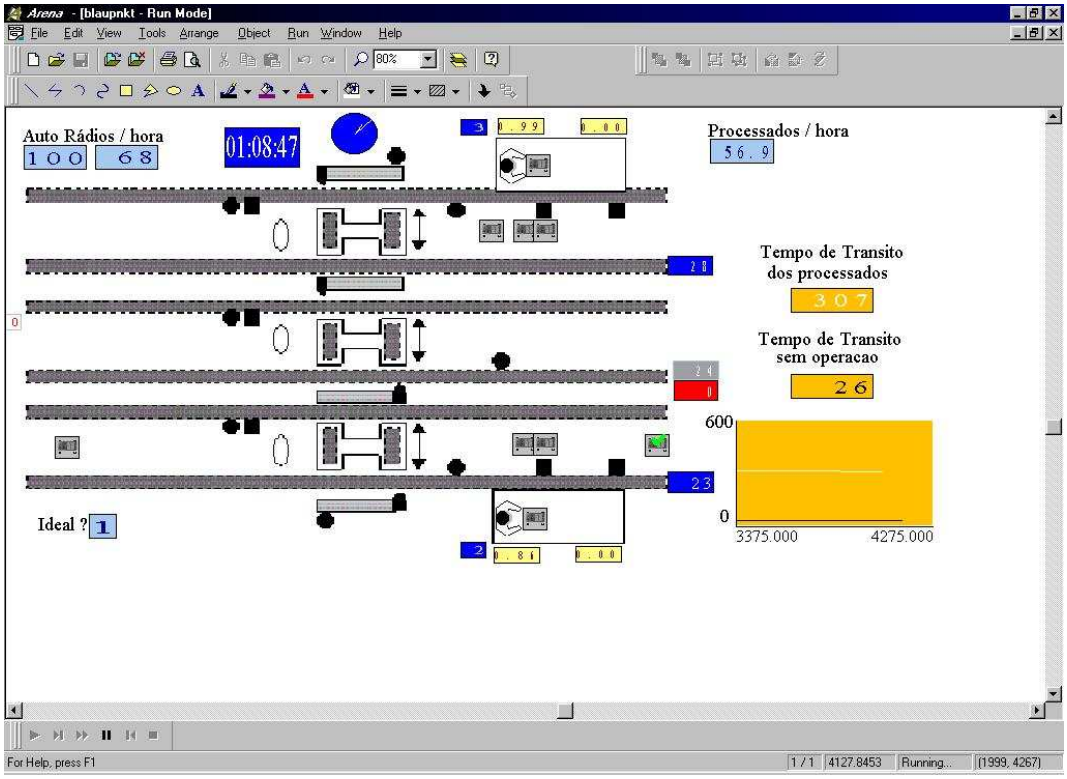


Figura 7.22: Aspecto da interface gráfica da aplicação de simulação em ARENA.

divisão do controlo em 2 níveis, feita na subsecção 7.2.8.

O sistema controlado duma linha actualmente em funcionamento na BLAUPUNKT (linha HIDRO #1) encontra-se abaixo descrita, usando uma notação textual, com a finalidade de simplificar a sua representação. A sintaxe usada recorre ao material apresentado no apêndice B e à caracterização da linha HIDRO #1 [Team BP-UM, 1999b, cap. 3, pág. 18–39]. Os elementos do sistema controlado (sensores e actuadores) são manipulados pelo sistema de controlo para conseguir pôr em prática o funcionamento pretendido.

$$controlado = \mathbf{NCES1I1}_0 + \sum_{n\in\{1,16\frac{1}{2}\}} \mathbf{NCES3I2}_n + \sum_{n\in\{2-6,8-14,16\}} \mathbf{NCS3P}_n + \sum_{n\in\{7,15\}} \mathbf{NCES3I2P}_n + \sum_{li\in\{A-E\}} \mathbf{m}_{c,li}$$

$$\mathbf{NCES1I1}_0 = [\mathbf{NBS}_{B,0} - (\mathbf{s}_{B,0,u} + \mathbf{i}_{B,0,u} + \mathbf{b}_{B,0}) - (\mathbf{s}_{B,0,r} + \mathbf{i}_{B,0,r,x} + \mathbf{i}_{B,0,r,y}) + \mathbf{o}_h + \mathbf{c}_h + \mathbf{s}_h + \mathbf{i}_h + \mathbf{i}_{B,0\frac{1}{3},p} + \mathbf{i}_{B,0\frac{2}{3},p}]+$$
$$[\mathbf{NBI}_{E,0} - (\mathbf{o}_{E,0} + \mathbf{b}_{E,0}) - (\mathbf{s}_{E,0,l} + \mathbf{i}_{E,0,l,x} + \mathbf{i}_{E,0,l,y}) - \mathbf{i}_{E,0,d} + \mathbf{i}_{E,0\frac{2}{3},u}] + \mathbf{NB}_{e,0}$$

$$\mathbf{NCES3I2}_1 = \mathbf{NCES3I2P}_n \mid_{n=1} -[(\mathbf{i}_{A,1,u} + \mathbf{s}_{A,1,u} + \mathbf{i}_{A,1,l,x}) + (\mathbf{i}_{C,1,u} + \mathbf{s}_{C,1,u}) + (\mathbf{s}_{C,1,r} + \mathbf{i}_{C,1,r,x})]-$$
$$[(\mathbf{o}_{D,1} + \mathbf{s}_{D,1,l} + \mathbf{i}_{D,1,l,x} + \mathbf{s}_{D,1,r} + \mathbf{i}_{D,1,r,x}) + (\mathbf{b}_{E,1} + \mathbf{i}_{E,1,r,y})] + (\mathbf{i}_{D,1\frac{1}{2},u} + \mathbf{s}_{D,1\frac{1}{2},u})$$

$$\mathbf{NCS3P}_{\{2-6,8-13,16\}} = \mathbf{NCS3P}_n - (\mathbf{i}_{A,n,p} + \mathbf{i}_{C,n,p})$$

$$\mathbf{NCES3I2P}_7 = \mathbf{NCES3I2P}_n \mid_{n=7} +[-(\mathbf{i}_{A,7,p} + \mathbf{i}_{C,7,p}) + \mathbf{s}_{A,7\frac{1}{3},p}] + [-(\mathbf{s}_{D,7,l} + \mathbf{i}_{D,7,l,x}) - (\mathbf{0}_{E,7} + \mathbf{b}_{E,7}) + (\mathbf{s}_{E,7\frac{1}{3},u} + \mathbf{s}_{E,7\frac{2}{3},u} + \mathbf{i}_{E,6\frac{1}{2},d})]$$

$$\mathbf{NCS3P}_{14} = \mathbf{NCS3P}_n \mid_{n=14} -\mathbf{i}_{A,14,p} + \mathbf{b}_{C,14\frac{1}{2}}$$

$$\mathbf{NCES3I2P}_{15} = \mathbf{NCES3I2P}_n \mid_{n=15} -(\mathbf{i}_{A,15,p} + \mathbf{i}_{C,15,p}) - [(\mathbf{o}_{D,15} + \mathbf{b}_{D,15}) + \mathbf{o}_{E,15}]$$

$$\mathbf{NCES3I2}_{16\frac{1}{2}} = \mathbf{NCES3I2}_n \mid_{n=16\frac{1}{2}} +[-(\mathbf{i}_{A,16\frac{1}{2},l,y} + \mathbf{i}_{C,16\frac{1}{2},l,x}) - (\mathbf{i}_{A,16\frac{1}{2},d} + \mathbf{i}_{B,16\frac{1}{2},d} + \mathbf{i}_{C,16\frac{1}{2},d}) + (\mathbf{i}_{B,16\frac{1}{2},p} + \mathbf{s}_{B,16\frac{1}{2},p})]+$$
$$[-(\mathbf{i}_{D,16\frac{1}{2},u} + \mathbf{s}_{D,16\frac{1}{2},u} + \mathbf{o}_{D,16\frac{1}{2}} + \mathbf{b}_{D,16\frac{1}{2}} + \mathbf{i}_{D,16\frac{1}{2},r,x} + \mathbf{s}_{D,16\frac{1}{2},r}) + \mathbf{i}_{D,16\frac{1}{2},l,y} - (\mathbf{b}_{E,16\frac{1}{2}})]$$

$$\mathbf{NCS3P}_n = \mathbf{NCS3}_n + \mathbf{NP}_n$$

$$\mathbf{NCS3}_n = \sum_{li\in\{A,B,C\}} \mathbf{NBS}_{li,n} - (\mathbf{s}_{A,n,r} + \mathbf{i}_{A,n,r,x} + \mathbf{i}_{A,n,r,y}) - (\mathbf{i}_{B,n,l,y} + \mathbf{i}_{B,n,r,y}) - (\mathbf{s}_{C,n,l} + \mathbf{i}_{C,n,l,x} + \mathbf{i}_{C,n,l,y})$$

$$\mathbf{NBS}_{li,n} = \mathbf{i}_{li,n,u} + \mathbf{s}_{li,n,u} + \mathbf{b}_{li,n} + (\mathbf{s}_{li,n,l} + \mathbf{i}_{li,n,l,x} + \mathbf{i}_{li,n,l,y}) + \mathbf{t}_{li,n} + \mathbf{m}_{t,li,n} + (\mathbf{s}_{li,n,r} + \mathbf{i}_{li,n,r,x} + \mathbf{i}_{li,n,r,y}) + \mathbf{i}_{li,n,d}$$

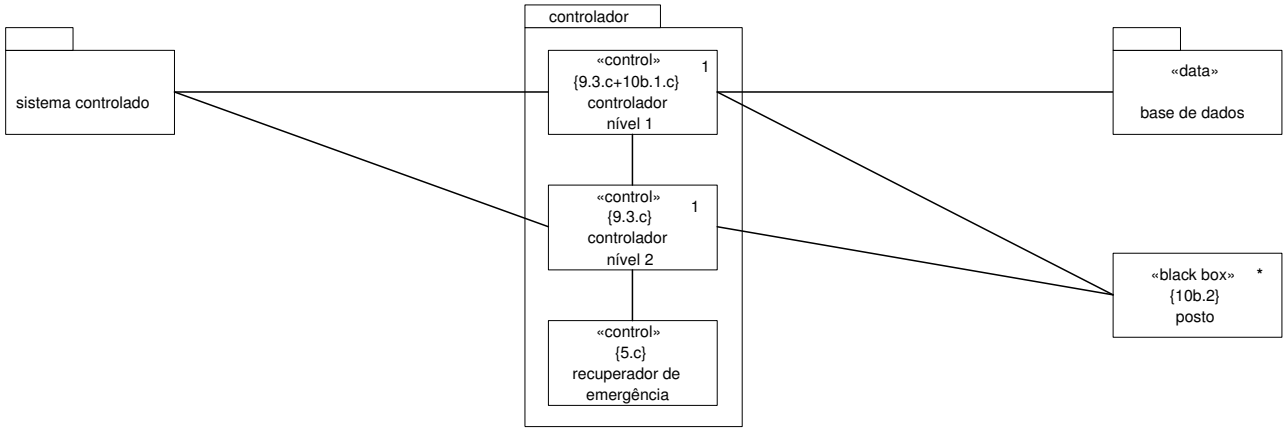


Figura 7.23: Diagrama final de objectos para o SCLH.

$$\mathbf{NP}_n = (\mathbf{P}_{n,l} + \mathbf{i}_{A,n,p} + \mathbf{s}_{A,n,p}) + (\mathbf{P}_{n,r} + \mathbf{i}_{C,n,p} + \mathbf{s}_{C,n,p})$$

$$\mathbf{NCES3I2P}_n = (\mathbf{NCS3P}_n - \mathbf{i}_{C,n,r,y} + \mathbf{i}_{C,n,l,x}) + (\mathbf{NCI2}_n - \mathbf{i}_{D,n,l,y}) + \mathbf{NB}_{e,n}$$

$$\mathbf{NCI2}_n = \sum_{li \in \{D,E\}} \mathbf{NBI}_{li,n} - \mathbf{i}_{D,n,r,y} - (\mathbf{s}_{E,n,l} + \mathbf{i}_{E,n,l,x} + \mathbf{i}_{E,n,l,y})$$

$$\mathbf{NBI}_{li,n} = \mathbf{NBS}_{li,n} + \mathbf{o}_{li,n}$$

$$\mathbf{NB}_{e,n} = \mathbf{e}_n + \mathbf{m}_{t,e,n} + \mathbf{i}_{e,n,s} + \mathbf{i}_{e,n,i} + \mathbf{i}_{e,n,j}$$

$$\mathbf{NCES3I2}_n = \mathbf{NCES3I2P}_n - \mathbf{NP}_n$$

7.2.10 Diagrama de classes

Na fig. 7.24 apresenta-se o diagrama de classes para o SCLH. Este diagrama organiza os módulos básicos de controlo (de nível 2) segundo uma hierarquia que coloca as classes mais simples (e mais abstractas) no topo e que recorre exclusivamente ao mecanismo de herança para relacionar as diversas classes.

Os nomes dados às classes seguiram um esquema que adere às seguintes regras:

- **ContN** para indicar que a classe diz respeito a um controlador dum nó do sistema controlado.
- **B** para controladores de nós básicos e **C** para controladores de nós compostos.
- **E** se o respectivo nó tem um elevador (só válido para nós compostos). Neste caso, pode ainda acrescentar-se **d**, **u** e **du**, se o elevador, tendo no seu interior auto-rádios, se movimenta, respectivamente, para baixo (*down*), para cima (*up*), e para baixo e para cima.
- **Sn** se o número de linhas no plano superior do respectivo nó é igual a n (no diagrama, n tomou os valores 1, 2, 3).
- **Im** se o número de linhas no plano inferior do respectivo nó é igual a m (no diagrama, m tomou os valores 1, 2, 3).
- **P** se o respectivo nó inclui postos.

7.2.11 Cenários de funcionamento

As várias regras definidas na subsecção 7.2.8 consubstanciam as estratégias de acesso aos transfers, de modo a evitar potenciais acidentes (por exemplo, choques de auto-rádios) e pretendem

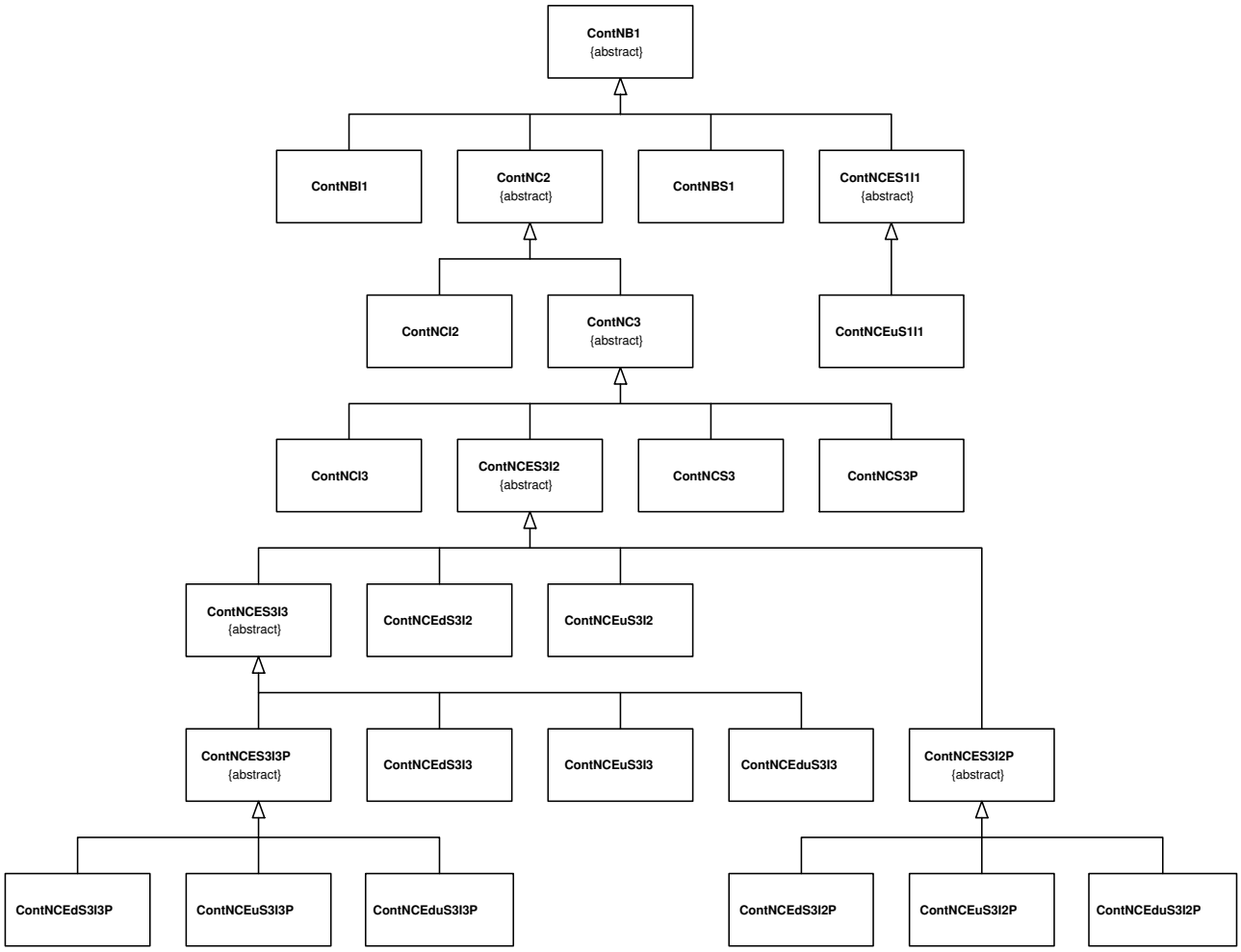


Figura 7.24: O diagrama de classes para o SCLH.

também indicar os procedimentos que dotam o sistema dos mecanismos que o permitam recuperar de situações imprevistas (por exemplo, ficar *ad eternum* à espera dum auto-rádio que foi retirado à mão). No nível 2 de controlo, caso não seja possível enviar um dado auto-rádio para o destino pretendido, pode alterar-se este com o intuito de não bloquear o nó em causa.

Nesta subsecção, apresentam-se 7 cenários, entendidos como sequências temporais de imagens instantâneas dum dado nó composto, que ilustram situações típicas de funcionamento das linhas, e tecem-se alguns comentários relativos a essas situações, tentando explicar as regras que estiveram na origem de determinados comportamentos.

Neste projecto, optou-se por representar os cenários de funcionamento usando uma notação facilmente entendida pelo cliente, em detrimento dos diagramas que UML disponibiliza para o efeito (diagramas de sequência e de colaboração). Trata-se duma forma muito útil de verificar, junto do cliente, se as estratégias de controlo adoptadas são ou não as mais adequadas. Refira-se que inicialmente foi usada notação UML (diagramas de sequência), mas o cliente mostrou dificuldades em perceber a relação entre as mensagens que fluem entre objectos e as implicações que aquelas têm na movimentação das paletes. Daí o recurso à alternativa escolhida, que pode sem grandes problemas ser transformada em diagramas de sequência [Team BP-UM, 1999a, pág. 41–2].

Cenário 1

No cenário da fig. 7.25, pode verificar-se que a palete #1 não avançou, no instante T1, devido à presença da palete #3 a jusante do transfer C. No instante T2, a palete #2 pôde avançar, pois todo o caminho que pretende percorrer estava livre. Depois da palete #2 ter sido encaminhada para o seu destino, no instante T5, a palete #1 já pode avançar, pois todo o caminho a percorrer está agora livre, visto que a palete #3 foi, entretanto, removida (entre os instantes T3 e T4).

Ao assumir-se este comportamento, evita-se que um auto-rádio seja enviado para os transfers, se é possível determinar, à partida, que algum dos troços, por onde ele vai passar, está ocupado. Trata-se duma forma preventiva de actuar, garantindo-se assim que não se vai bloquear o nó ou agravar uma situação já de si bloqueada e permitindo que, eventualmente, outros auto-rádios possam utilizar o nó em causa, antes do auto-rádio anteriormente considerado o fazer.

Cenário 2

Considere-se agora o cenário da fig. 7.26, em que se verifica que, entre os instantes T2 e T3, apareceu indevidamente a palete #3 no transfer C, barrando assim o caminho à palete #1. No instante T3, o destino desta palete é alterado, passando a ser a linha $\mathbf{L}_B$.

Neste caso usou-se a linha $\mathbf{L}_B$, como alternativa, pois não foi possível enviar o auto-rádio que já estava nos transfers para o destino, devido à palete #3 estar no transfer C. Deste modo, evita-se que um auto-rádio fique parado nos transfers, o que a acontecer iria bloquear o nó em causa, até o destino de nível 2, para esse auto-rádio, estar livre. Com a aceitação deste comportamento, garante-se que os auto-rádios têm sempre um destino de saída, mesmo naquelas situações em que, por alguma razão, o respectivo caminho ficou impedido. Se a linha $\mathbf{L}_B$ também está bloqueada, então o nó ficará bloqueado até a linha B ficar desimpedida. No plano inferior, o comportamento é idêntico, mas é usada a linha $\mathbf{L}_E$ como alternativa

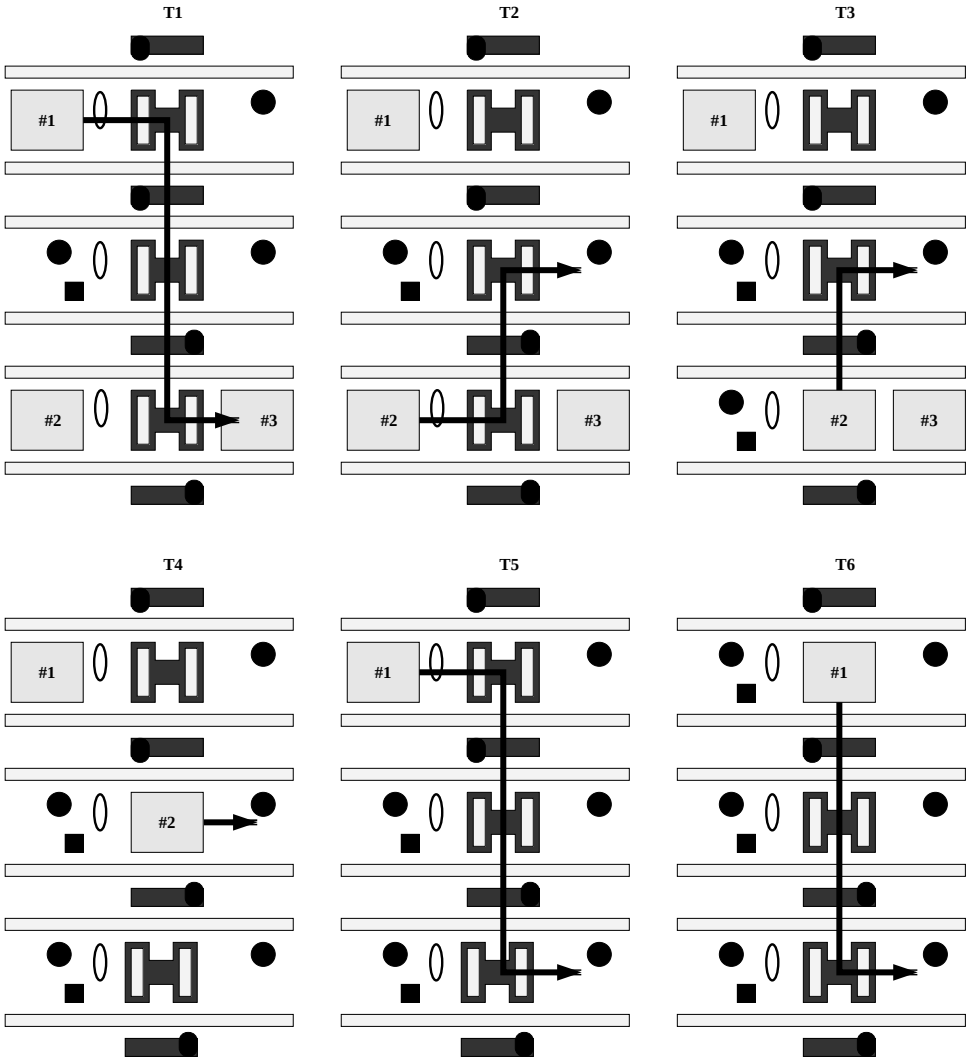


Figura 7.25: Cenário 1: regra ree-7.

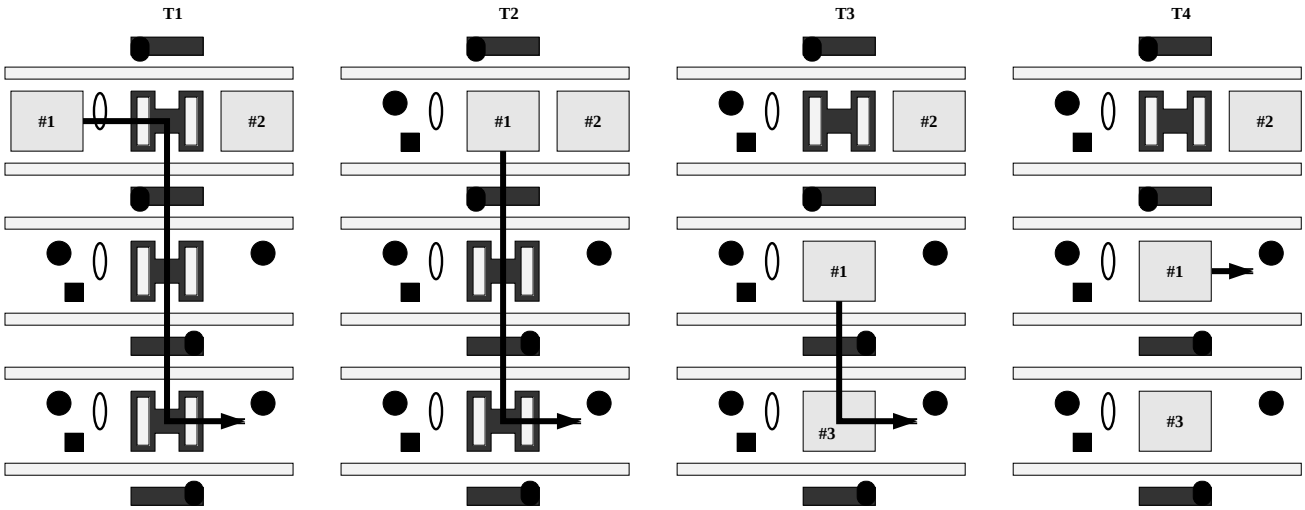


Figura 7.26: Cenário 2: regras rtf-5.ii e rod-2.

Cenário 3

O cenário da fig. 7.27 pretende mostrar que, antes dum auto-rádio iniciar um movimento elementar, deve confirmar-se se ele ainda lá se encontra. Um *movimento elementar* é definido como o percurso mais pequeno que se pode observar nas linhas Hidro. Relativamente ao cenário da fig. 7.26, existem 3 movimentos elementares: de T1 para T2 (carregar o transfer A), de T2 para T3 (deslocar para o transfer à direita) e de T4 para T5[1] (descarregar o transfer B).

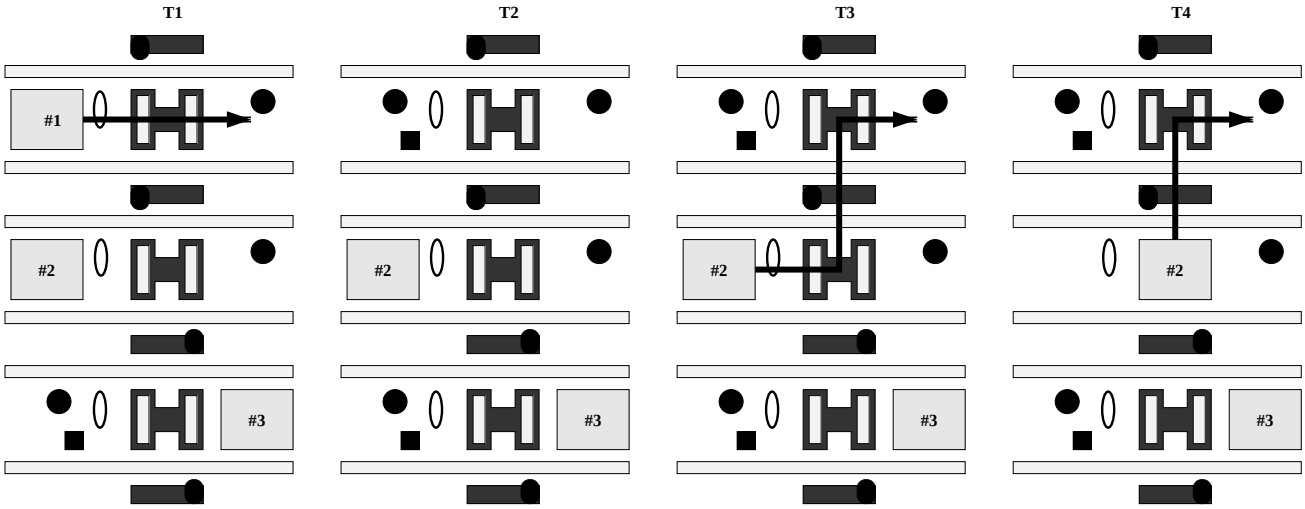


Figura 7.27: Cenário 3: regra rtf-5.i ou rtf-5.iii.

Retomando o cenário da fig. 7.27, verifica-se que, no instante T2, desapareceu a palete #1 (admita-se que alguém a retirou manualmente). O sistema não deve ficar à espera que o auto-rádio volte a aparecer no local donde foi retirado, podendo focar-se no tratamento de outro

[1]Apesar do instante T5 não estar indicado na figura, supõe-se que a seguir a T4 a palete #1 será enviada em frente, para a linha $\mathbf{L}_B$.

auto-rádio (palete #2 nos instantes T3 e T4). Este cenário mostra que se podem detectar todas as situações em que alguém retirou manualmente, junto aos transfers, um auto-rádio antes de o movimentar (rtf-5.i) ou depois do movimento ter sido iniciado (rtf-5.iii). Desta forma, é escusado ficar indefinidamente à espera que o auto-rádio apareça na origem ou no destino, pois há a certeza que ele foi retirado da linha.

Cenário 4

O cenário da fig. 7.28 mostra que, no instante T2, apareceu "misteriosamente" a palete #3 (admita-se que alguém a colocou lá manualmente), barrando assim o caminho à palete #1. Só após a palete #3 desaparecer, pode a palete #1 seguir para o destino pretendido. Neste exemplo, considerou-se que foi possível, dentro dum dado intervalo temporal, enviar a palete para o seu destino, pois, caso tal não sucedesse, deveria, por aplicação da regra rod-2, ter-se alterado o destino da palete #1 para $\mathbf{L}_B$ (fig. 7.31).

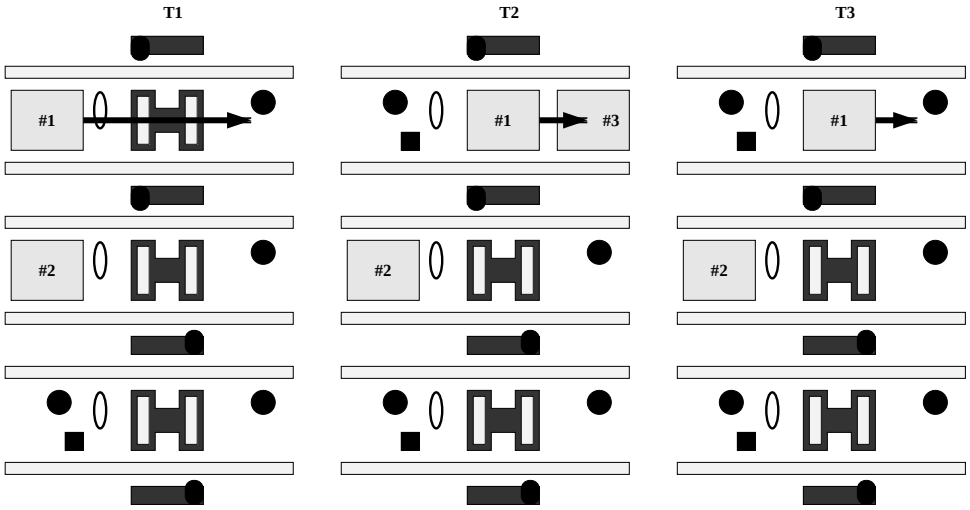


Figura 7.28: Cenário 4: rtf-5.ii.

Do cenário da fig. 7.28, pode concluir-se que, antes de um auto-rádio iniciar um movimento elementar, deve verificar-se se está livre o local onde termina esse movimento. Assim, evitam-se choques entre auto-rádios e podem detectar-se todas as situações em que auto-rádios são colocados, manualmente, junto aos transfers.

Cenário 5

Relativamente ao cenário da fig. 7.29, constata-se que, no instante T1, apareceu inesperadamente a palete #2 (admita-se que foi lá colocada à mão por alguém), barrando o caminho à palete #1. O tratamento desta situação obriga a enviar a palete #2 para a linha $\mathbf{L}_B$, podendo depois retomar-se o trajecto inicialmente previsto para a palete #1, como se mostra nos instantes T3 e T4.

Generalizando, pode afirmar-se que se um auto-rádio está no transfer A (ou C) e tem de ser movimentado para o transfer B, que está inesperadamente ocupado, então há que, primeiramente, libertar o transfer B, enviando o auto-rádio que lá está para a frente. Quando tal suceder,

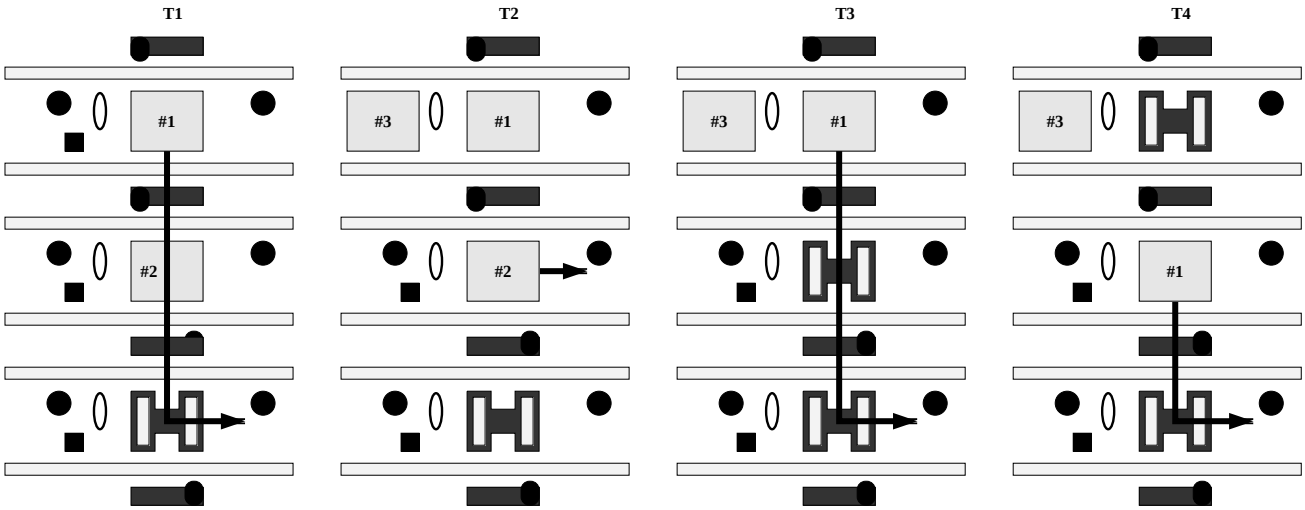


Figura 7.29: Cenário 5: rtf-5.ii e rod-2.

o auto-rádio original deve ser tratado novamente. Desta forma, evita-se que se dêem choques entre auto-rádios e podem detectar-se todas as situações em que auto-rádios são colocados no transfer B.

Cenário 6

Quanto ao cenário da fig. 7.30, pode observar-se que, no instante T1, apareceu inesperadamente a palete #2 (assuma-se que foi lá colocada manualmente por alguém), barrando o caminho à palete #1. O tratamento desta situação implica, primeiramente, o envio da palete #1 para a linha $\mathbf{L}_B$ (instante T2) e, depois, da palete #2 também para a linha $\mathbf{L}_B$ (instantes T3 e T4).

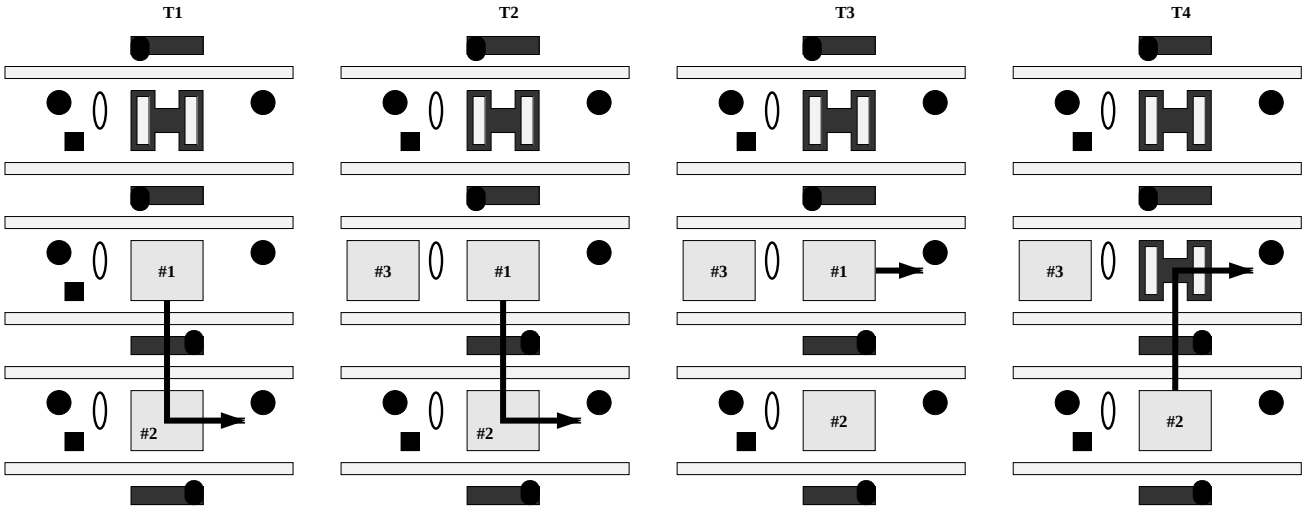


Figura 7.30: Cenário 6: rtf-5.ii e rod-2.

Conclui-se, pois, que se um auto-rádio está no transfer B e tem de ser movimentado para o

transfer C (ou A), que está ocupado, então há que enviar esses dois auto-rádios para jusante do transfer B, i.e. para a linha $\mathbf{L}_B$.

Com este procedimento, além de se resolver a situação descrita, evitam-se choques entre auto-rádios e detectam-se todas as situações em que auto-rádios são colocados nos transfers A ou C. Como não se sabe qual é o destino do auto-rádio que apareceu no transfer C (ou A), ao enviá-lo para a linha $\mathbf{L}_B$, não se corre o risco de estar a enviar esse auto-rádio para um posto. A penalização reside no facto do auto-rádio, que originalmente se pretendia enviar para o transfer C (ou A), ser enviado para a frente, o que significa que terá que fazer, eventualmente, um ciclo (via um elevador), caso não existam, mais postos a jusante que realizem a mesma tarefa que aquela que era realizada no posto das linhas $\mathbf{L}_C$ (ou $\mathbf{L}_A$).

Outra solução que se equacionou, mas que não se considerou correcto adoptar, foi a de enviar ambos os auto-rádios para a linha $\mathbf{L}_C$ (ou $\mathbf{L}_A$). Neste caso, porém, corria-se o risco de os transfers ficarem bloqueados, caso o *buffer* da linha $\mathbf{L}_C$ (ou $\mathbf{L}_A$) a montante do nó estivesse cheio. Adicionalmente, podia suceder que o auto-rádio que surgiu no transfer C (ou A) fosse incorrectamente enviado para o posto a jusante do transfer da linha $\mathbf{L}_C$ (ou $\mathbf{L}_A$), caso não fosse esse o seu destino.

Cenário 7

O último cenário (fig. 7.31) mostra que, no instante T1, apareceu inesperadamente a palete #2 (suponha-se que foi lá colocada), barrando o caminho à palete #1. A forma como esta situação é resolvida consiste em enviar a palete #1 para a linha $\mathbf{L}_B$ (instantes T3 e T4).

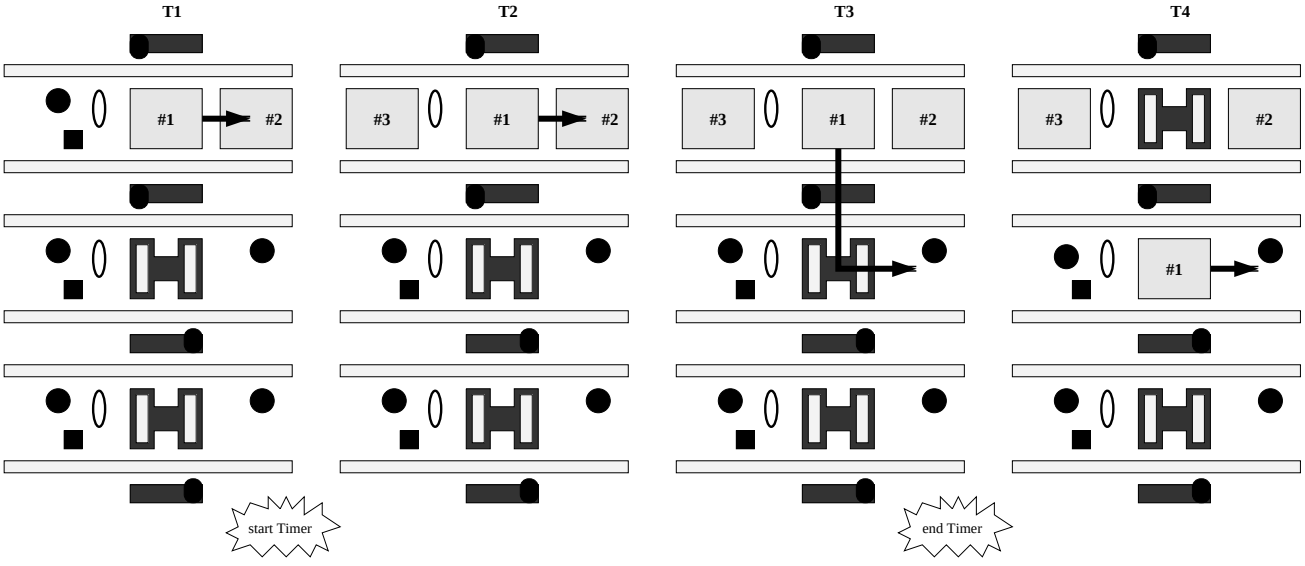


Figura 7.31: Cenário 7: rtf-5.ii e rod-2.

Se um auto-rádio está no transfer A (ou C) e tem de ir em frente, mas não o pode fazer porque o *buffer* a jusante do respectivo transfer está cheio, deve aguardar-se um determinado tempo pré-definido (*time-out*), na expectativa de que esse *buffer* fique com espaço livre. Quando esse tempo se esgotar, se existir um outro auto-rádio à entrada do transfer e se o *buffer* continuar cheio, o primeiro auto-rádio deve ser encaminhado para a linha $\mathbf{L}_B$, já que está a impedir a

circulação duma outra palete.

Deste modo, evita-se que um auto-rádio fique muito tempo à espera nos transfers, até que um dado *buffer* fique livre, o que, caso não sucedesse, poderia impedir outros auto-rádios de se movimentarem. Repare-se que foi a presença da palete #3, a montante do transfer A, que desencadeou o envio da palete #1 para a linha $\mathbf{L}_B$. A utilização dos temporizadores permite verificar periodicamente se há auto-rádios a montante do transfer que ficou bloqueado. Se a palete #3 tivesse aparecido a montante dos transfers B ou C, não haveria necessidade de enviar a palete #1 para a linha $\mathbf{L}_B$. Nessa situação, se o destino da palete #3 for as linhas $\mathbf{L}_B$ ou $\mathbf{L}_C$, a palete #1 não impede esse movimento; se a linha $\mathbf{L}_A$ for o destino da palete #3, esta deve ser reencaminhada para a linha $\mathbf{L}_B$, pois o *buffer* da linha $\mathbf{L}_A$ está cheio[2].

7.2.12 Diagramas de state-charts

São apresentados nas fig. 7.32–7.46 os state-charts que, em conjunto, definem o comportamento dinâmico dum controlador (de nível 2) de um nó composto superior com 3 linhas (e sem elevador). Estes 15 state-charts definem o comportamento dos objectos da classe ContNCS3 da fig. 7.24. Em alternativa, poderia ter-se optado por definir um único state-chart, mas a solução escolhida permite dividir funcionalmente o sistema (em sub-sistemas mais simples), o que consiste numa forma conhecida de atacar a complexidade dos sistemas. Além disso, permite que alguns dos state-charts possam ser directamente reaproveitados para outras classes, uma vez que a sua construção já foi feita com esse cuidado.

Nos state-charts apresentados, surgem diversas chamadas a rotinas que permitem definir completamente o comportamento do controlador considerado (parte do objecto 9.3.c). Essas rotinas podem ser interpretadas, segundo o paradigma dos objectos, como sendo operações disponíveis em objectos que constituem o SCLH. Algumas das operações pertencem ao próprio controlador em causa e outras pertencem a objectos com que esse controlador comunica. Os quadros 7.8 e 7.9 listam essas operações e apresenta uma pequena descrição para cada uma delas. Durante a modelação, a construção dos state-charts e da descrição das rotinas foi feita em paralelo, devendo, assim que estiverem concluídos, retomar-se o diagrama de classes e incluir, na zona das operações das classes afectadas, todas as rotinas que são referidas nos state-charts.

Em relação a algumas condições do quadro 7.9, deve notar-se que freeTr = ¬busyTr, freeDn = ¬busyDn, Up = ¬Dn, Superior = ¬Inferior. Relativamente a um nó, admitiu-se ainda que as linhas e o elevador, quando existir, constituem um conjunto totalmente ordenado. Em termos matemáticos, pode afirmar-se que o conjunto das linhas $\mathbf{L}_A \ldots \mathbf{L}_F$ e do elevador e constitui um ordem total.

$$\mathbf{L}_A < \mathbf{L}_B < \mathbf{L}_C < e < \mathbf{L}_D < \mathbf{L}_E < \mathbf{L}_F$$

e que estão definidas duas operações sobre as linhas (soma e subtracção), como a seguir se indica:

$$\mathbf{L}_A + 1 = \mathbf{L}_B \qquad\qquad \mathbf{L}_B - 1 = \mathbf{L}_A$$

[2]Alternativamente, poderia enviar-se primeiramente a palete #1 para a linha $\mathbf{L}_B$ e depois tentar enviar a palete #3 para a linha $\mathbf{L}_A$. Como há fortes probabilidades do *buffer* desta linha continuar cheio, a palete #3 ficaria à espera de espaço livre. Na prática, o resultado seria muito idêntico: uma palete enviada para o nó seguinte e outra à espera que o *buffer* ficasse com espaço.

Operação	**Descrição**
ReadBar(li,n)	Devolve o valor lido pelo código de barras que está a montante do transfer *li,n* ($\mathbf{b}_{li,n}$). Pode falhar, entre outros, por a etiqueta do código estar mal colada, por a palete estar vazia ou por problemas de comunicação.
Dest(li,n,id,calc)	Pede à base de dados (via controlador de nível 1) que lhe indique qual o destino de nível 1 do auto-rádio com o código *id* localizado no transfer *li,n*. Pode falhar por a base de dados ou o controlador de nível 1 estarem em baixo, por erros em operações anteriores ou por problemas de comunicação. O parâmetro *calc* determina o tipo de cálculo pretendido, no âmbito das regras ree-6, rod-2 e rae-2.
DefDest(li,n)	Determina o destino duma palete, caso tenha falhado o cálculo. Para linhas superiores, o destino será a linha $\mathbf{L}_B$ ou, se existir, uma via rápida, enquanto que para as linhas inferiores, o destino será a linha $\mathbf{L}_E$.
Msg(li,n,ref)	Regista informação de log, relativa a alguma situação anómala sucedida na vizinhança do transfer *li,n*.
Go(li,n,pos,dir)	Faz deslocar a palete que está na vizinhança do transfer *li,n*, no sentido indicado por *dir* (#F em frente, #R à direita, #L à esquerda). O parâmetro *pos* indica a posição da palete em relação ao tranfer (#Up a montante, #Tr no transfer) antes do deslocamento.
Stop(li,n,pos,dir)	Pára os meios mecânicos que permitiram mover a palete recorrendo à operação Go(li,n,pos,dir).
Change(n,idx,dst)	Verifica, quando idx for uma linha do meio (idx=B) ou uma via rápida, se está livre o destino final do movimento (no caso, o buffer a jusante do transfer dst, n). Se tal não se verificar, o destino final deve ser alterado para o valor da linha actual (idx) onde se encontra o auto-rádio.
CallElev(li,n)	Movimenta o elevador $\mathbf{e}_n$ para o plano adequado (i.e. o plano da linha $\mathbf{L}_l i$) antes do envio da palete para o interior do elevador (supostamente para o enviar para uma linha do outro plano).
Elev(n,sns)	Movimenta o elevador $\mathbf{e}_n$ no sentido sns. O parâmetro sns pode ter os valores Up ou Down, consoante o movimento se faça no sentido ascendente ou descendente, respectivamente.
CalcSense(dst)	Determina o sentido do movimento necessário para realizar o movimento para o destino dst. Se o destino pertencer ao nível superior, o resultado é up. Caso contrário, o resultado é down.

Tabela 7.8: As operações necessárias para completar o comportamento especificado nos statecharts.

Operação	**Descrição**
busyUp(li,n)	Devolve TRUE se o sensor $\mathbf{i}_{li,n,u}$ estiver On, o que permite determinar se existe uma palete a montante do transfer *li,n*.
busyTr(li,n)	Dependendo da linha *li*, devolve TRUE se um dos sensores $\mathbf{i}_{li,n,r}$ ou $\mathbf{i}_{li,n,l}$ estiver On, de modo a determinar se o transfer *li,n* está ocupado.
busyDn(li,n)	Devolve TRUE se o sensor $\mathbf{i}_{li,n,d}$ estiver On, o que possibilita determinar se o *buffer* a jusante do transfer *li,n* está cheio ou se uma palete está a passar, nesse instante, por cima do sensor.
freeTr(li,n)	Dependendo da linha *li*, devolve TRUE se um dos sensores $\mathbf{i}_{li,n,r}$ ou $\mathbf{i}_{li,n,l}$ estiver Off, permitindo detectar se o transfer *li,n* está livre.
freeDn(li,n)	Devolve TRUE se o sensor $\mathbf{i}_{li,n,d}$ estiver Off. Permite determinar se o *buffer* a jusante do transfer *li,n* está livre.
Superior(li)	Devolve TRUE se a linha $\mathbf{L}_{li}$ for do plano superior, i.e. $li \in \{A, B, C\}$.
Inferior(li)	Devolve TRUE se a linha $\mathbf{L}_{li}$ for do plano inferior, i.e. $li \in \{D, E, F\}$.
Optic(li,n)	Devolve TRUE se o sensor $\mathbf{o}_{li,n}$ estiver On, o que indica a existência de um auto-rádio em cima da palete.
SameLevel(li,dst)	Devolve TRUE se as linhas $\mathbf{L}_{li}$ e $\mathbf{L}_{dst}$ forem do mesmo plano. Considera-se que se uma das linhas for e (elevador), o resultado é TRUE, pois o elevador tanto pode estar em baixo como em cima.
Comp(sns,n)	Devolve TRUE se o sentido sns do movimento pretendido puder ser realizado pelo elevador $\mathbf{e}_n$. Embora, os elevadores possam fazer transportes verticais nos dois sentidos, consideram-se apenas os sentidos para os quais o elevador foi programado fazer transportes de paletes.
UpDown(n)	Devolve TRUE se o elevador $\mathbf{e}_n$ estiver programado para se movimentar nos 2 sentidos (para cima e para baixo), com paletes no seu interior.
Up(e,n)	Devolve TRUE se o sensor $\mathbf{i}_{e,s}$ estiver a On, o que indica que o elevador $\mathbf{e}_n$ está no plano superior.
Dn(e,n)	Devolve TRUE se o sensor $\mathbf{i}_{e,i}$ estiver a On, o que indica que o elevador $\mathbf{e}_n$ está no plano inferior.
Priority(li,n)	Devolve TRUE se o troço da linha $\mathbf{L}_{li}$ entre os nós n e $n+1$ for uma via rápida ou pertencer à linha $\mathbf{L}_B$ ou $\mathbf{L}_E$.
TestTrack(li,n,dst)	Devolve TRUE se, no nó n, estiver livre o percurso entre o transfer da linha $\mathbf{L}_{li}$ e o *buffer* a montante do transfer $\mathbf{L}_{dst}$.

Tabela 7.9: As condições necessárias para completar o comportamento especificado nos statecharts.

$$\begin{array}{ll} \mathbf{L}_B + 1 = \mathbf{L}_C & \mathbf{L}_C - 1 = \mathbf{L}_B \\ \mathbf{L}_C + 1 = e & e - 1 \quad = \mathbf{L}_C \\ e + 1 \quad = \mathbf{L}_D & \mathbf{L}_D - 1 = e \\ \mathbf{L}_D + 1 = \mathbf{L}_E & \mathbf{L}_E - 1 = \mathbf{L}_D \\ \mathbf{L}_E + 1 = \mathbf{L}_F & \mathbf{L}_F - 1 = \mathbf{L}_E \end{array}$$

Relativamente à operação Msg, disponível no objecto C1 (controlador de nível 1), o 3^o parâmetro de entrada (ref) especifica qual a situação que sucedeu. As situações registadas podem referir-se a acontecimentos anómalos ou a movimentos elementares terminados com sucesso. Para cada um dos valores possíveis para esse parâmetro, que seguidamente se apresentam, indica-se o seu significado:

- #BarC: código de barras foi mal lido.
- #GetUp: auto-rádio foi retirado a montante do transfer.
- #GetTr: auto-rádio foi retirado do transfer.
- #GetEl: auto-rádio foi retirado do elevador.
- #PutTr: auto-rádio foi colocado no transfer.
- #PutDn: auto-rádio foi colocado a jusante do transfer.
- #Rst: foi necessário desbloquear o transfer.
- #Sense: sentido de movimento do elevador não compatível com o pretendido.
- #Elev: movimento para o elevador foi completado com sucesso.
- #Load: movimento para o transfer foi completado com sucesso.
- #Right: movimento para a direita foi completado com sucesso.
- #Left: movimento para a esquerda foi completado com sucesso.
- #Unload: movimento para jusante do transfer foi completado com sucesso.

De seguida, fazem-se alguns comentários à forma como os state-charts foram modelados, tentando chamar a atenção para os aspectos que se consideram mais relevantes e para alguns mecanismos de modelação.

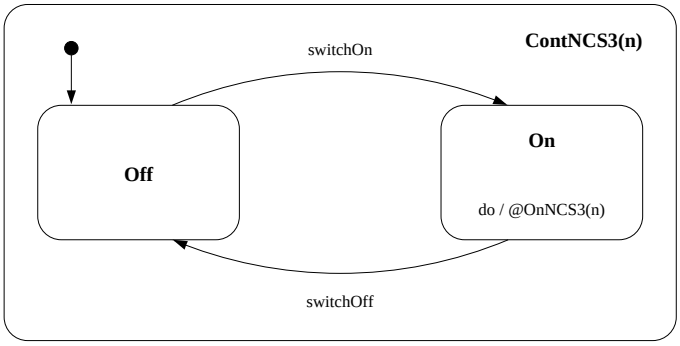


Figura 7.32: Statechart principal para objectos da classe ContNCS3.

No state-chart OperNCS3 (fig. 7.34), realce para as 3 regiões ortogonais, que permitem que em cada nó possam ser detectadas, em paralelo, 3 paletes (tantas quantas as linhas disponíveis). A gestão dos possíveis choques entre paletes é gerida no estado Destination do state-chart Proc (fig. 7.36), através da invocação do método Dest ao objecto C1 (controlador de nível 1). Este método, quando solicitado, é responsável por indicar o destino dos auto-rádios (valor

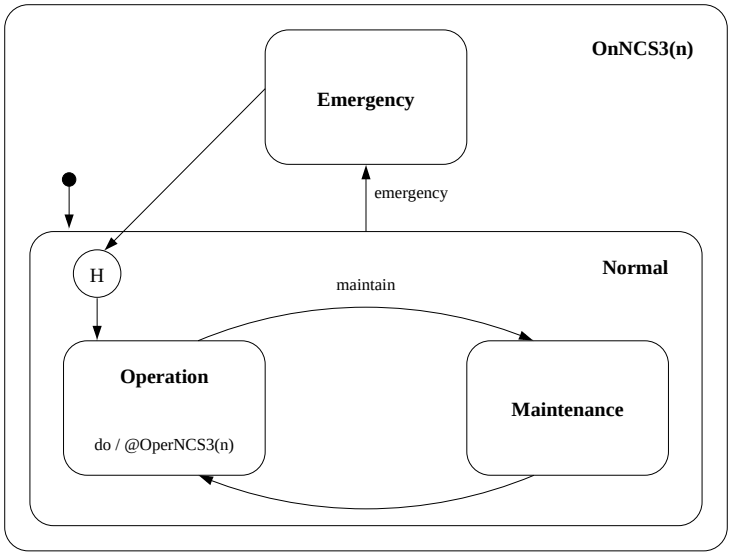


Figura 7.33: Statechart OnNCS3 para objectos da classe ContNCS3.

armazenado na variável dst) e ao fazê-lo está implicitamente a dar autorização ao controlo de nível 2 para fazer entrar esse auto-rádio no nó (cf. regras ree-6, ree-7, rat-1 e rat-2).

O 3^o parâmetro de entrada (calc) da função Dest permite que o controlo de nível 2 informe o controlo de nível 1 do tipo de cálculo pretendido:

- `calc = #First`: o controlador de nível 1 deve calcular o destino da palete no âmbito da regra ree-6;
- `calc = #Recalc`: o controlador de nível 1 deve calcular o destino da palete no âmbito da regra rod-2;
- `calc = #Elev`: o controlador de nível 1 deve calcular o destino da palete no âmbito da regra rae-2.

No state-chart ResetNCS3 (fig. 7.35), cuja função é limpar todos os transfers do nó n (leia-se enviar para a frente todas as paletes que estejam no nó, desempedindo assim este), a estratégia seguida reside em limpar primeiro o transfer B, depois o transfer A e finalmente o transfer C (poderia ter-se optado igualmente por inverter a ordem dos últimos 2 transfers). Repare-se na transição que inicia no contorno do state-chart e termina no estado CheckTB. Esta transição significa que, se em algum estado se verificar um erro, se deve voltar ao estado CheckTB. Trata-se duma medida de precaução, pois se, ao tentar limpar-se os transfers, houver algum problema, deve recomeçar-se tudo de novo.

A acção de entrada ret=OK, no estado CheckTB foi colocada, de modo a evitar que o state-chart ficasse indefinidamente retido nesse estado, pois a transição, com a condição [ret==ERROR], que termina nesse estado também se aplica a ele próprio. Caso não fosse alterado o valor de ret no estado CheckTB, aquela transição estaria sempre habilitada e o state-chart não estaria especificado duma forma determinística, pois uma segunda transição estaria igualmente habilitada (as condições busyTr e freeTr sao complementares).

Nos state-charts das fig. 7.35–7.46, note-se o pseudo-estado final, no fundo destes state-charts, que indica que o state-chart terminou e que se deve voltar ao state-chart que chamou este. É o equivalente a um retorno duma função (instrução `return` em linguagem C).

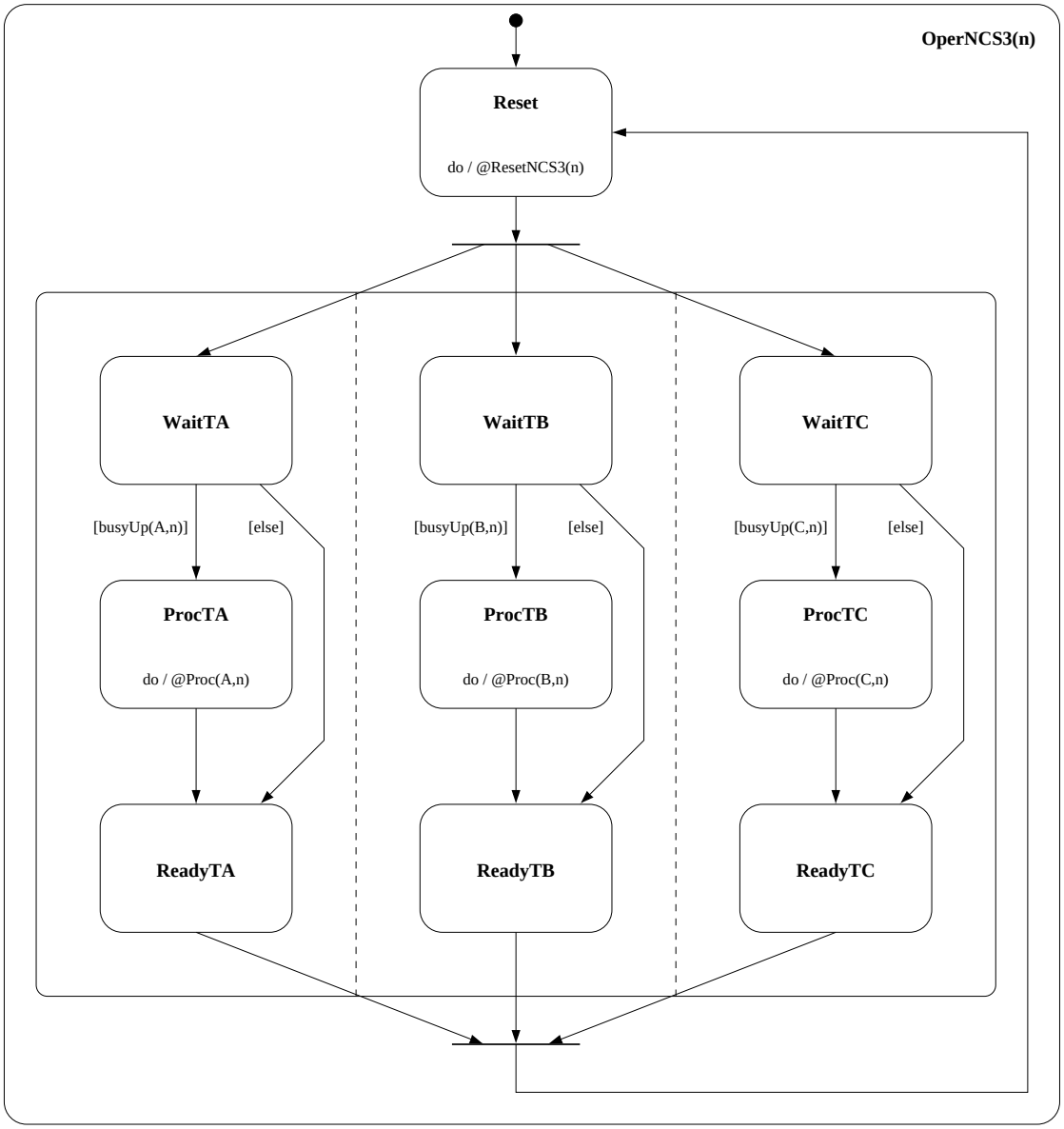


Figura 7.34: Statechart OperNCS3 para objectos da classe ContNCS3.

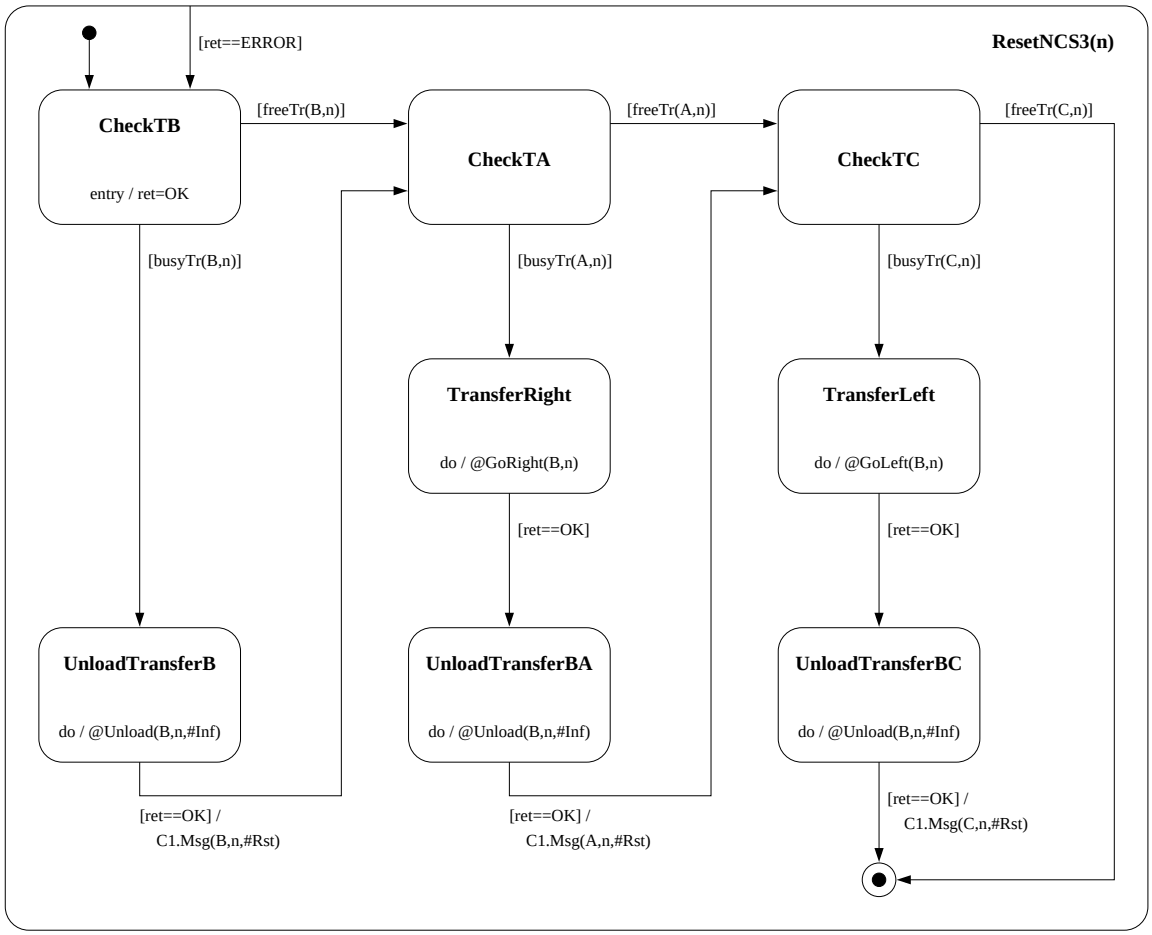


Figura 7.35: Statechart ResetNCS3 para objectos da classe ContNCS3.

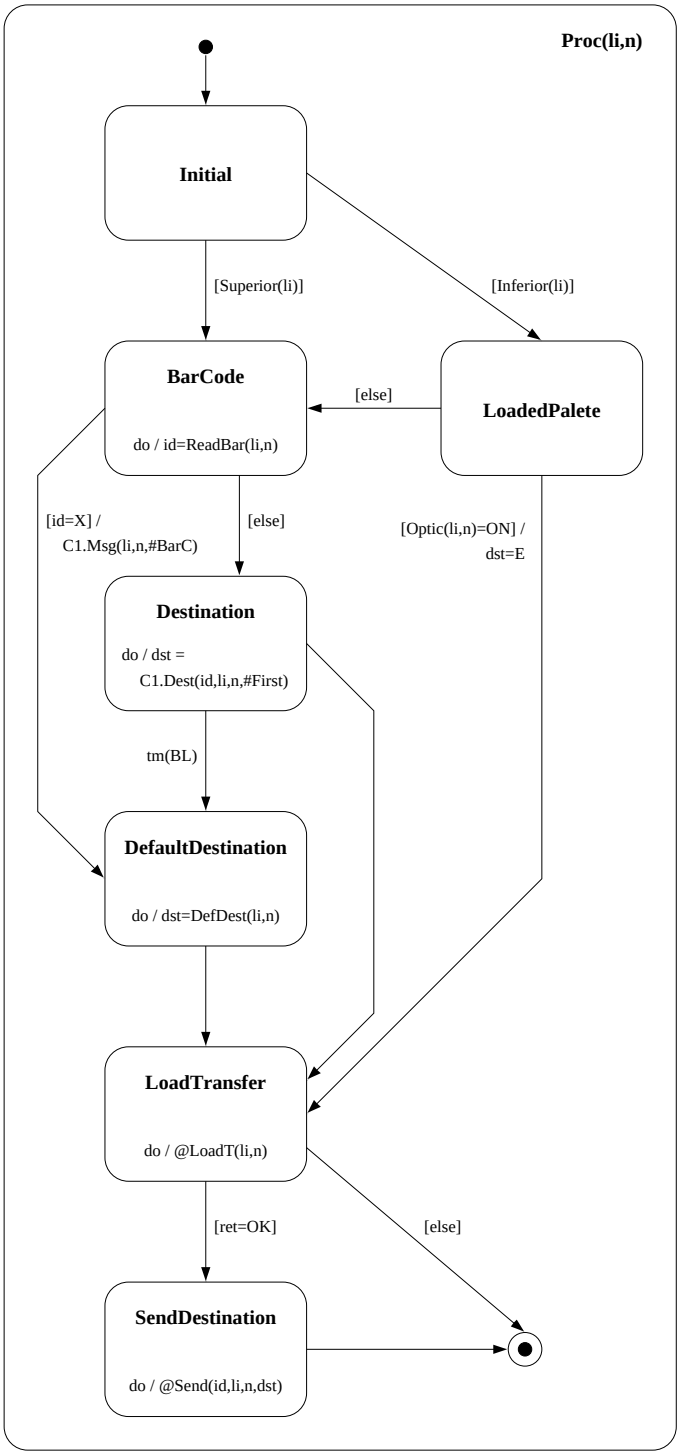


Figura 7.36: Statechart para descrever operação Proc.

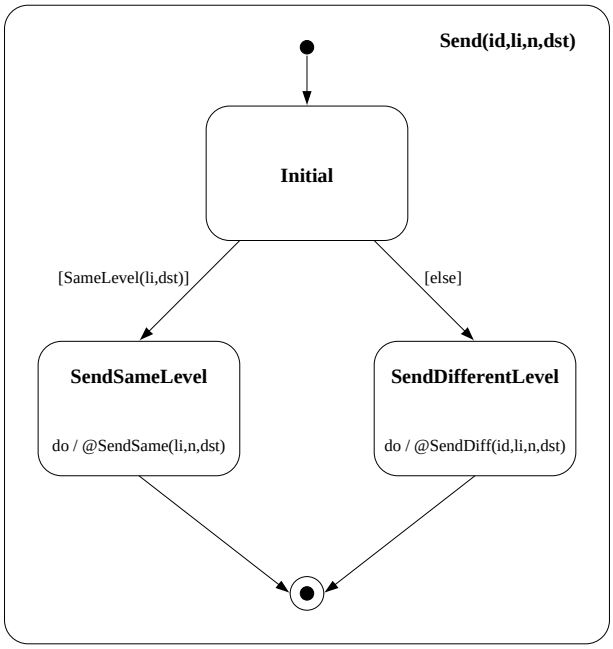


Figura 7.37: Statechart para descrever operação Send.

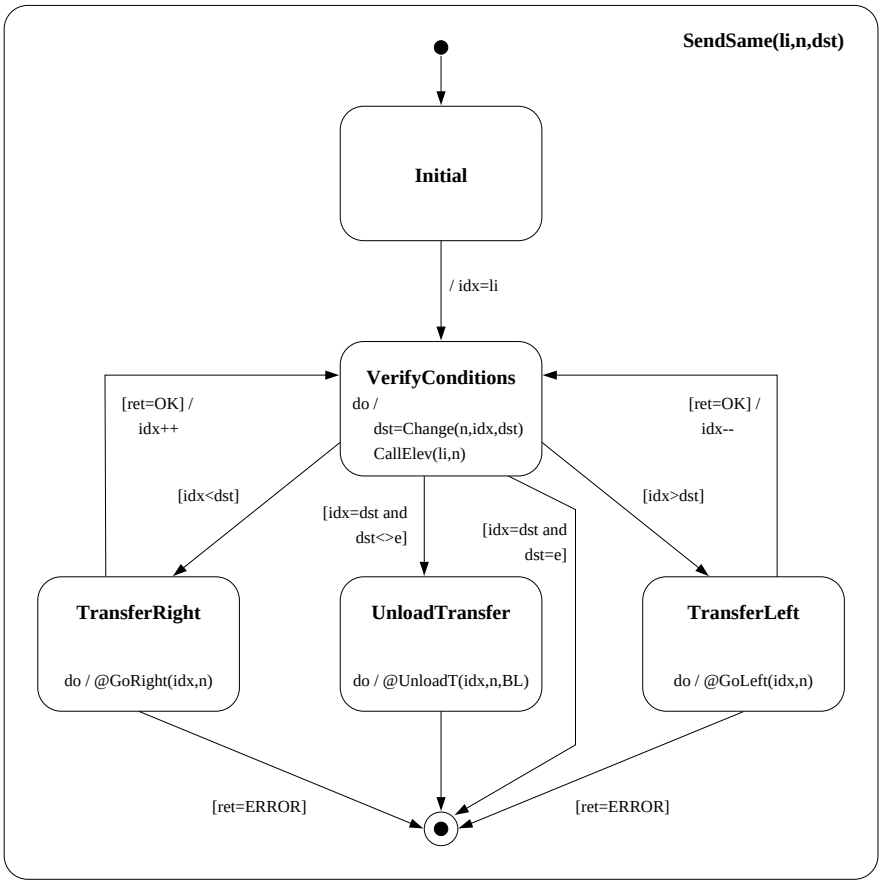


Figura 7.38: Statechart para descrever operação SendSame.

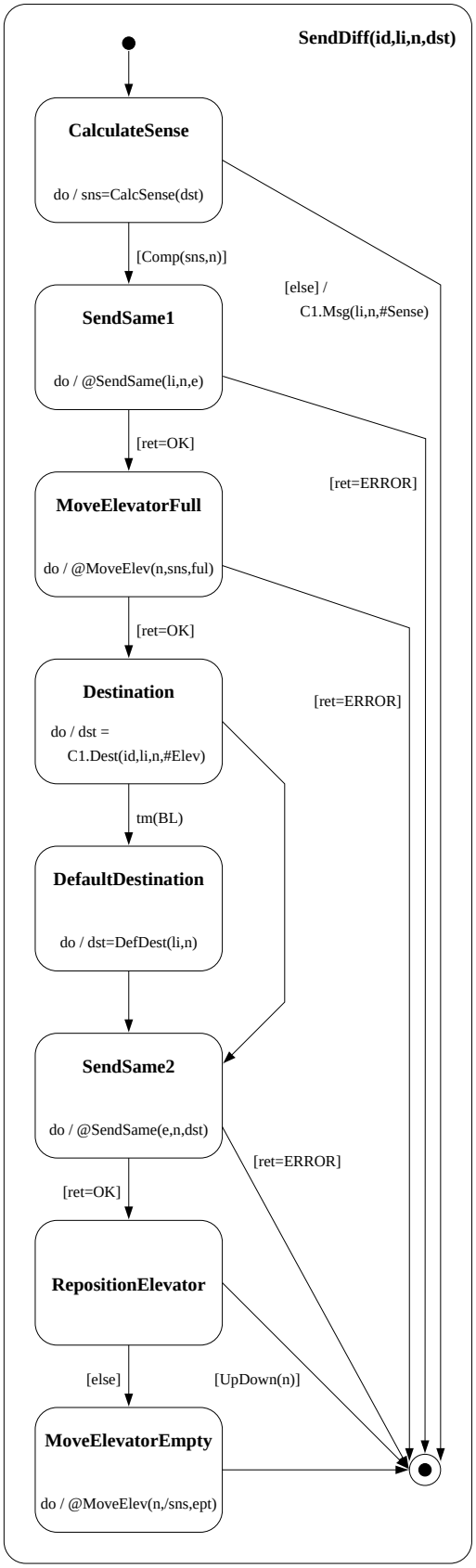


Figura 7.39: Statechart para descrever operação SendDiff.

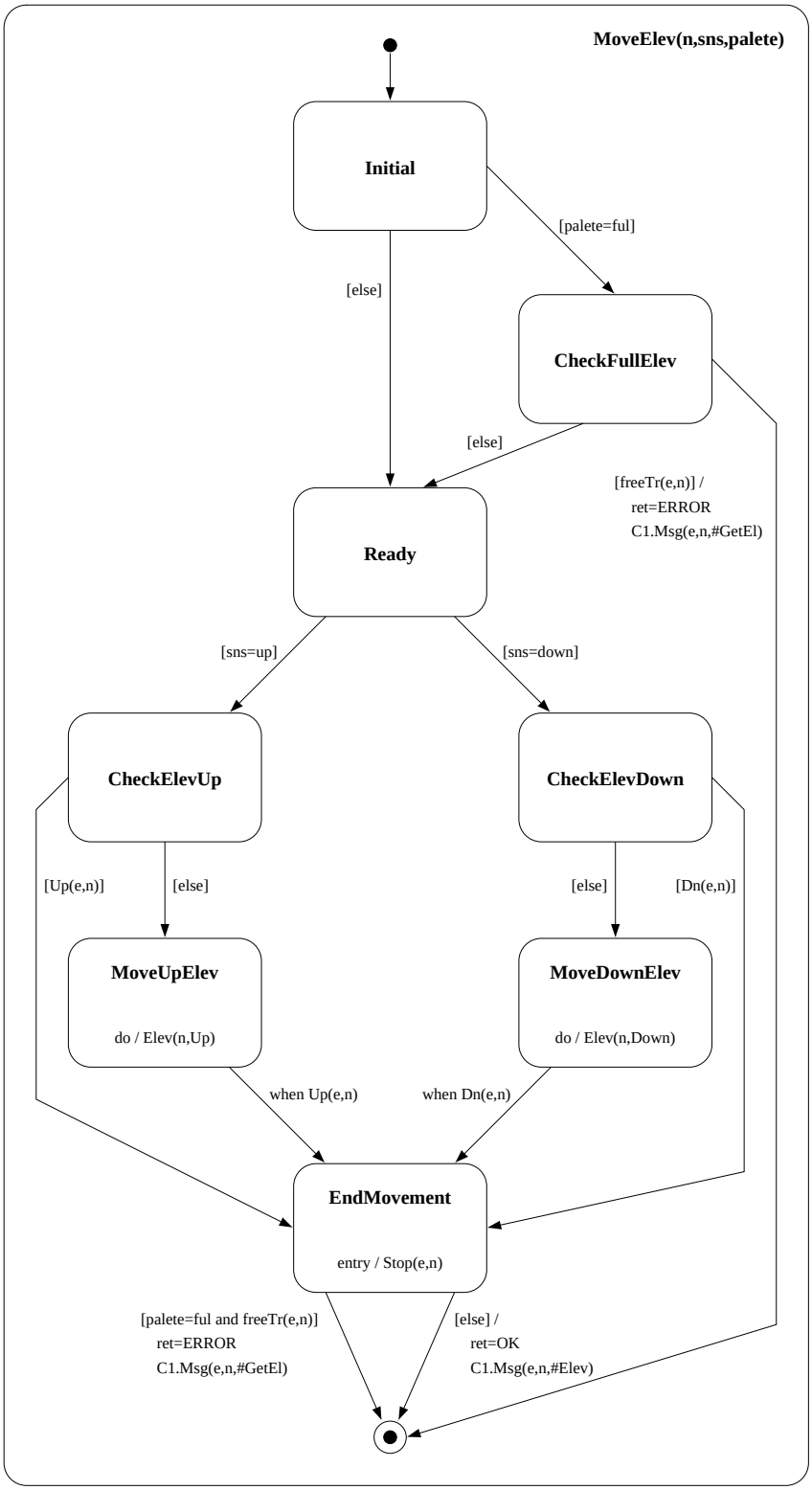


Figura 7.40: Statechart para descrever operação MoveElev.

Em relação aos state-charts das fig. 7.41–7.44, é de referir, o uso do evento temporal tm. Este evento especifica um prazo temporal e pressupõe a existência dum temporizador que dispara assim que passar o tempo indicado como parâmetro, após ter sido activado o estado donde provém a transição. O evento é cancelado se, entretanto, se abandonar o estado, por outra transição, antes do prazo imposto se verificar.

O estado CheckTransfer do state-chart LoadT (fig. 7.41) foi escolhido como exemplo, para explicar as razões pelas quais se recorreu a estes eventos temporizados na modelação destes state-charts. Após activar o estado CheckTransfer, o temporizador associado ao evento tm(BL) arranca. Se durante o tempo BL, a condição [freeTr(li,n)] não for verdadeira, disparará a transição que termina no pseudo-estado final. Caso contrário, dispara a transição que termina no estado MoveRadio. Esta modelação permite que o state-chart mude de estado, mesmo que o transfer *li,n* nunca fique livre, o que torna o sistema tolerante a faltas e permite detectar quando e onde se verificaram situações anómalas.

Note-se que qualquer das transições com temporizadores tem associada uma acção que coloca a variável ret com o valor ERROR. É nessas circunstâncias que a transição da sub-máquina da fig. 7.35 que tem associada a condição [ret==ERROR] dispara, independentemente do estado que estiver activo na altura. Retomando as fig. 7.41–7.44, pode observar-se que se não for disparada nenhuma transição com temporizador, então atinge-se o pseudo-estado final, o que faz que a acção ret=OK seja executada.

No state-chart Load (fig. 7.41) pode confirmar-se, caso haja um leitor de código de barras, se a identificação do auto-rádio que é detectado pelo sensor $\mathbf{i}_{li,n,u}$ ainda é o mesma que a inicialmente obtida. Este teste, a ser adicionado na transição que liga os estados CheckUp e CheckTransfer, permite detectar se, entre a primeira identificação e o início da movimentação do auto-rádio para o transfer, não houve qualquer manipulação estranha.

Nos state-charts GoRight e GoLeft (fig. 7.42 e 7.43), o parâmetro li representa a linha de destino do movimento. Assim, no state-chart GoRight, o valor $li - 1$ representa a linha origem (mais à esquerda) e no state-chart GoLeft, o valor $li + 1$ representa a linha origem (mais à direita). Estas informações estão reflectidas nas notas de texto colocadas no topo das figuras.

Para uma leitura mais simples do comportamento descrito pelos vários state-charts anteriormente apresentados, o quadro 7.10 indica, para cada uma das regras de controlo de nível 2, quais as partes dos state-charts que a põem em prática.

Como atrás já se referiu, a maioria dos state-charts apresentados pode ser directamente utilizada na especificação de outras classes que não apenas aquela que mereceu tratamento nesta secção (ContNCS3), uma vez que houve esse cuidado de generalização. Por exemplo, o state-chart Send (fig. 7.37) pode também ser usado para controladores de nós que incluam, simultaneamente, linhas superior e inferiores (ContNCS3I3). Assim sendo, seria conveniente que esse state-chart fosse especificado no contexto duma superclase de todas as classes que dele necessitam (por exemplo, ContNB1).

Outra hipótese teria sido especializar alguns state-charts, de forma a incluir apenas as partes relevantes para o tipo nó em causa. Por exemplo, para objectos da classe ContNCS3 (ContNCI3), seria conveniente, no state-chart Proc, alterar a actividade do estado SendDestination para SendSame, uma vez que, para esses nós, só podem existir movimentações no plano superior (inferior). Desta forma, o state-chart Send deixa de ser necessário para a classe ContNCS3. Para objectos da classe ContNCS3I3 e das respectivas subclasses, seria indispensável a versão do state-chart Proc, apresentada na fig. 7.36, dado ser possível observarem-se movimentações

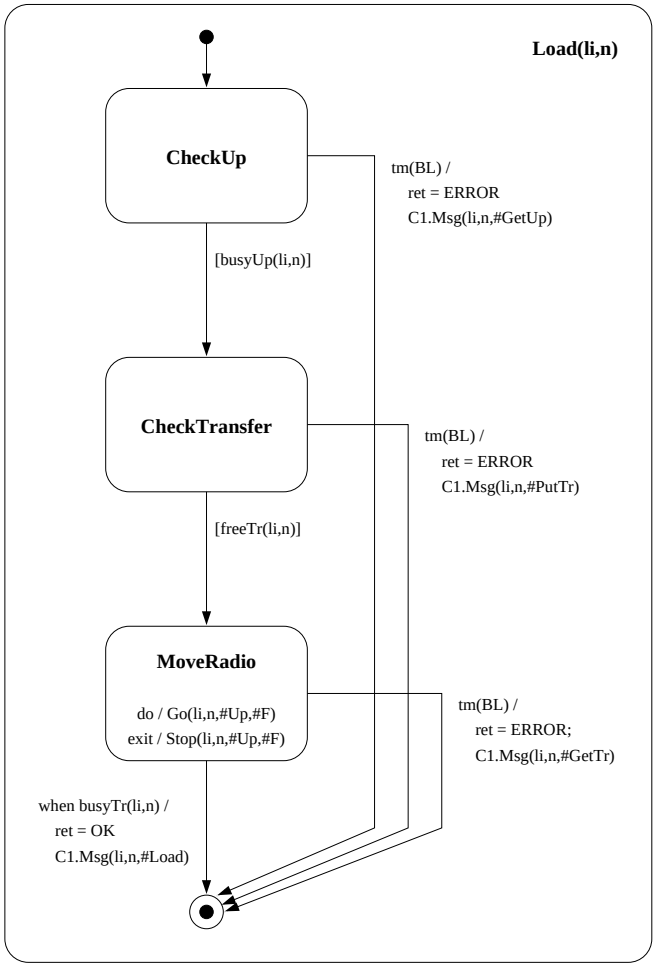


Figura 7.41: Statechart para descrever operação Load.

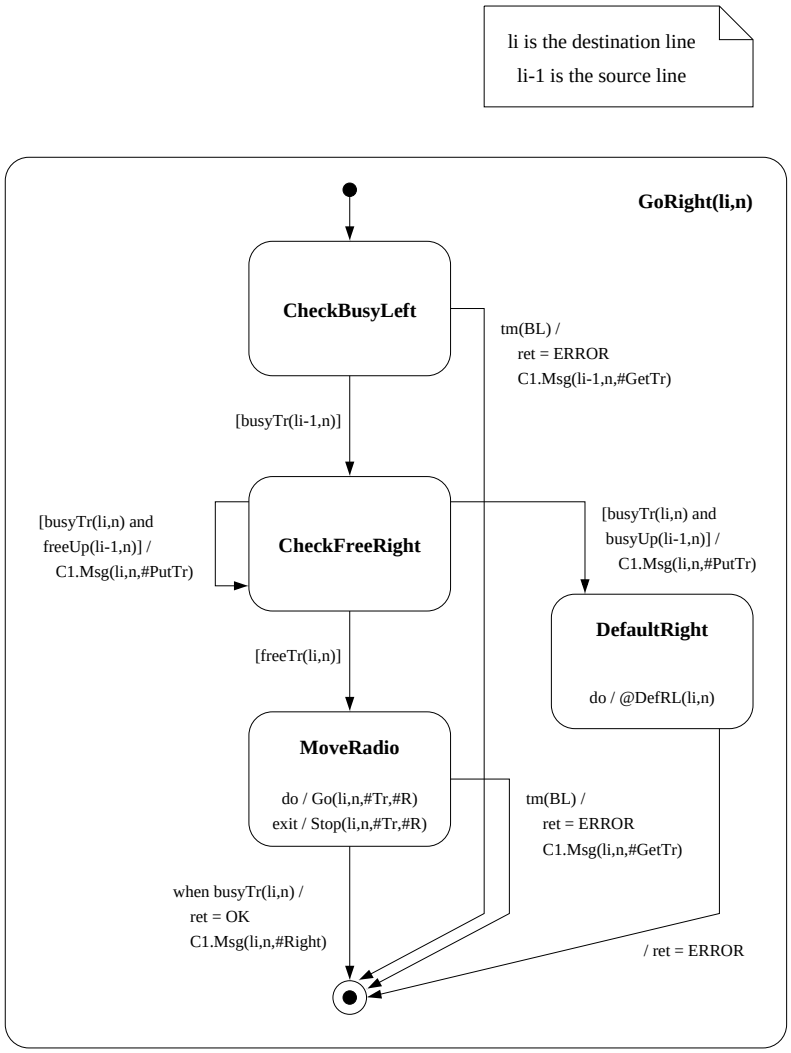


Figura 7.42: Statechart para descrever operação GoRight.

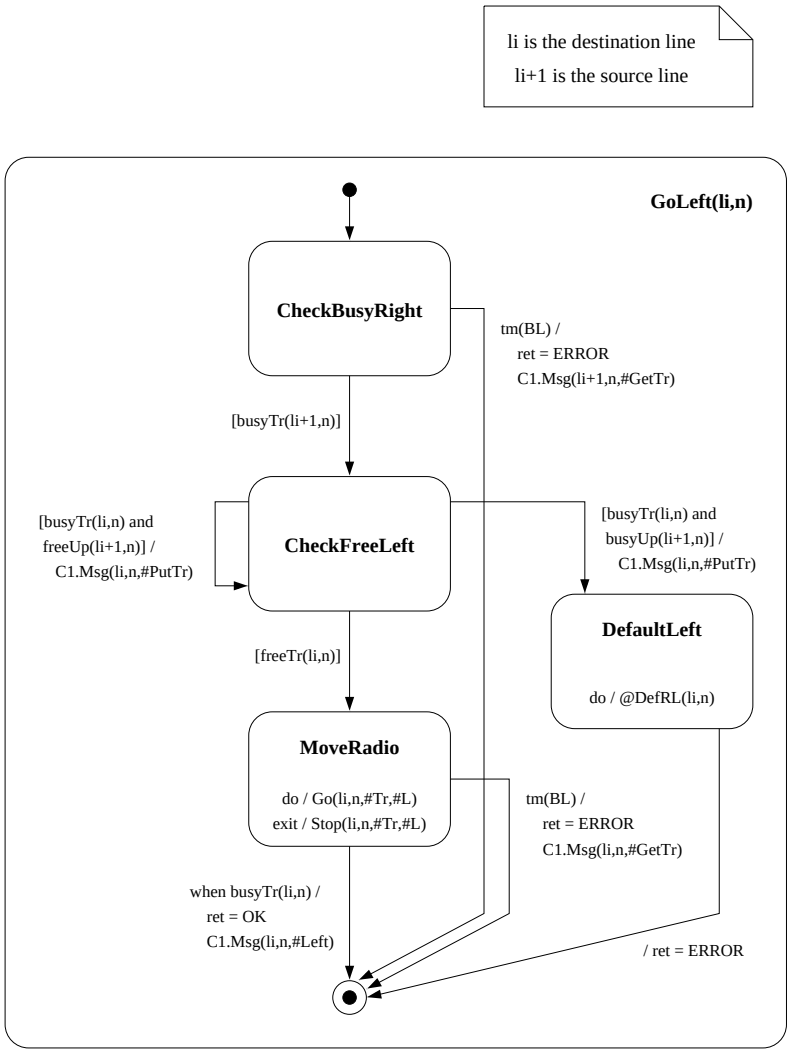


Figura 7.43: Statechart para descrever operação GoLeft.

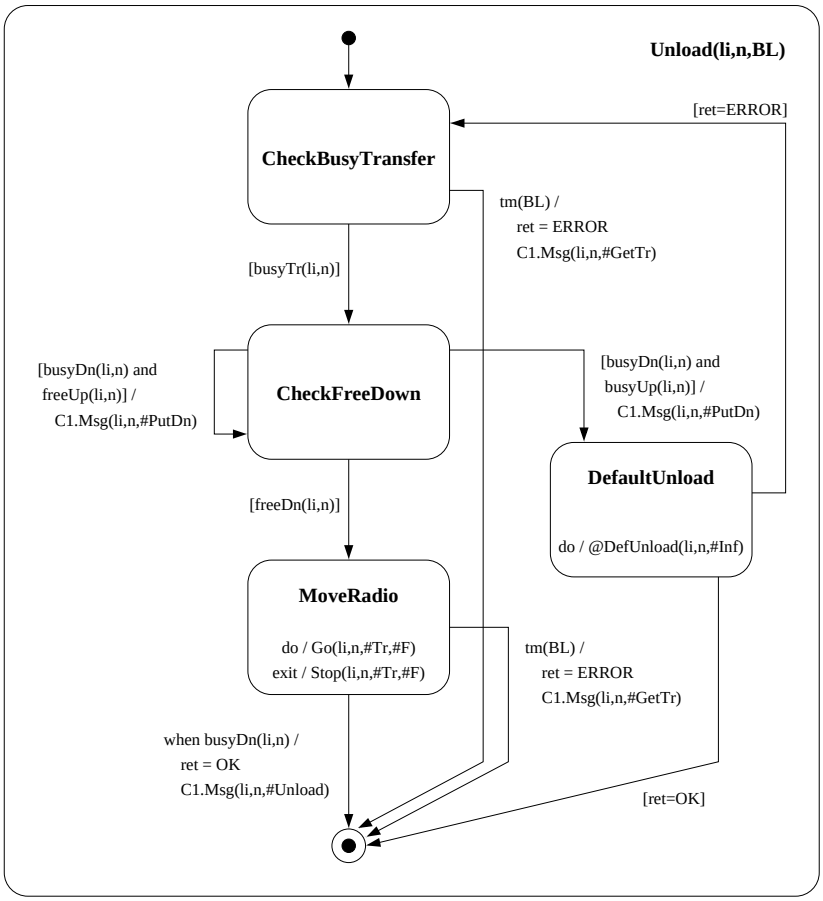


Figura 7.44: Statechart para descrever operação Unload.

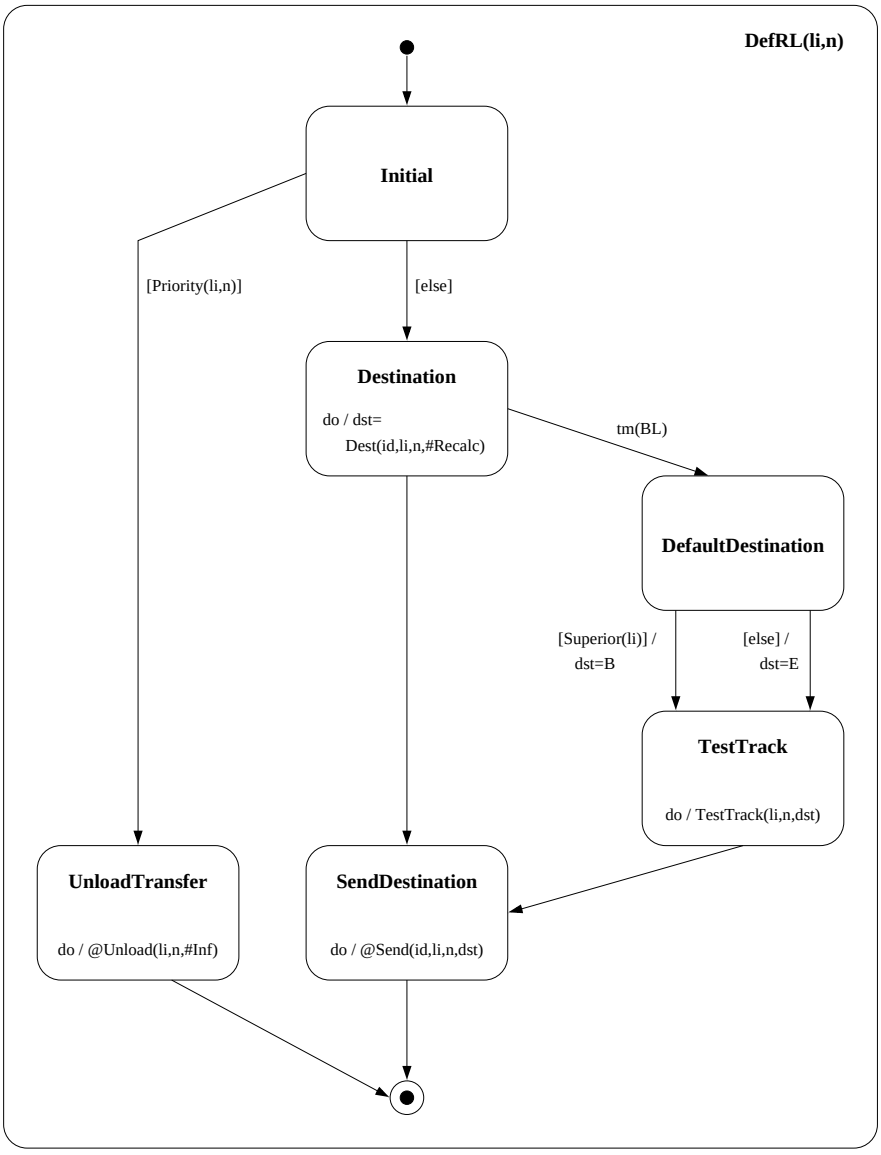


Figura 7.45: Statechart para descrever operação DefRL.

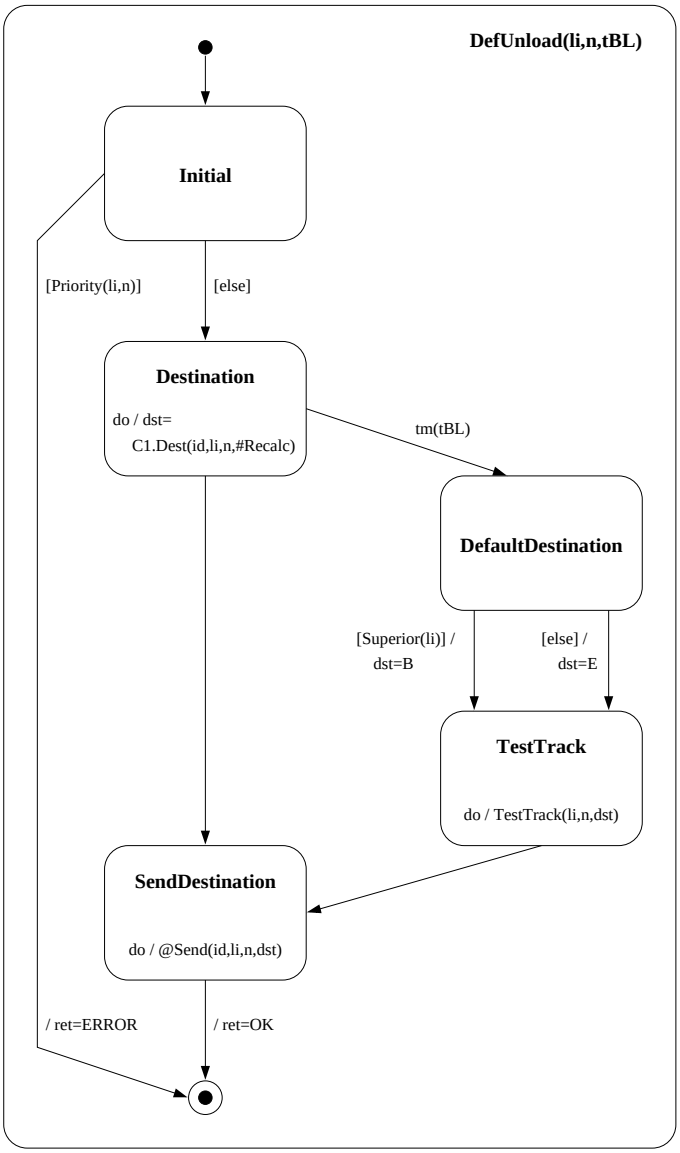


Figura 7.46: Statechart para descrever operação DefUnload.

Regra	State-charts
rat-1	OperNCS3: condições busyUp Proc: estados BarCode e Destination
rat-2	Proc: estado Destination
rat-3	OperNCS3: regiões ortogonais
rat-4	OperNCS3: estado Reset e transição junção
rae-1	Send: condição else (SameLevel=FALSE) SendDiff: estado MoveElevatorFull
rae-2	SendDiff
rtf-1	a tratar na implementação
rtf-2	Load, GoLeft, GoRight, Unload: eventos tm()
rtf-3	Proc, SendDiff: estados DefaultDestination
rtf-4	Proc: estado BarCode
rtf-5.i	Load: condição busyUp GoLeft, GoRight, Unload: condições busyTr
rtf-5.ii	Load, GoLeft, GoRight: condições freeTr Unload: condição busyDn
rtf-5.iii	Load, GoLeft, GoRight, Unload: eventos when(busyTr)
rtf-6	Vários: acções C1.Msg
rtf-7	Vários: acções C1.Msg
rod-1	Proc: condição id=X e estado DefaultDestination
rod-2	GoLeft: transições para DefaultLeft e própria GoRight: transições para DefaultRight e própria Unload: transições para DefaultUnload e própria
rod-3	SendSame: função Change

Tabela 7.10: Lista de regras de nível 2 e state-charts que as implementam.

entre o mesmo plano (comportamento especificado por SendSame) ou entre planos distintos (comportamento especificado por SendDiff).

Para o SCLH, optou-se pela generalização da maioria dos state-charts. Contudo, é necessário, para cada classe distinta, criar, como se fez para a classe ContNCS3, um state-chart equivalente a OperNCS3. Para a classe ContNCI3, esse state-chart seria OperNCI3 (fig. 7.47). Note-se que os state-charts ContNCS3 (fig. 7.32) e OnNCS3 (fig. 7.33) são genéricos para outras classes e não específicos para a classe ContNCS3, apesar dos nomes usados poderem indiciar apenas esta última hipótese.

Para descrever o comportamento dinâmico dos sistemas (ou de alguns dos seus componentes), recorreu-se aos state-charts, como atrás se apresentou, dado ser este o formalismo que a notação UML impõe. Contudo, no âmbito do projecto, foram paralelamente usadas RdP-shobi [Machado, 1996] e da comparação dos modelos que resultaram da adopção dos meta-modelos state-chart e RdP-shobi ressaltaram as seguintes conclusões:

1. Com state-charts não se consegue modelar paralelismo dinâmico, situação fácil e naturalmente modelada com RdP-shobi. O paralelismo dinâmico mostrou-se uma característica importante quando se pretendia modelar a possibilidade de até 3 auto-rádios poderem, em paralelo e no mesmo ciclo de controlo, usar o mesmo nó composto, desde que as suas

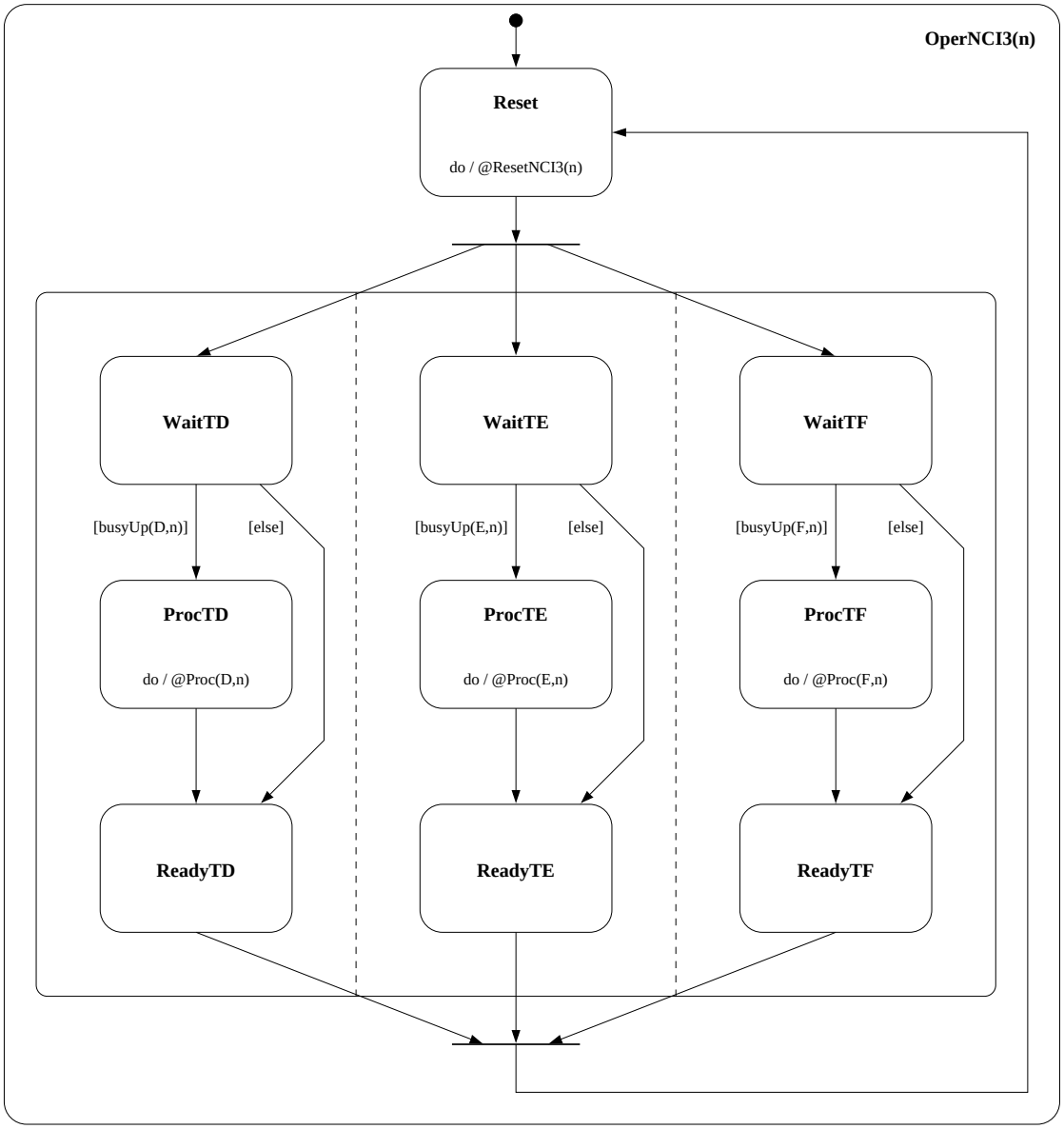


Figura 7.47: Statechart OperNCI3 para objectos da classe ContNCI3.

rotas não se cruzassem. O paralelismo dinâmico resulta do facto de o número de fios de controlo, a utilizar em cada ciclo, depender do número de auto-rádios detectados a montante do nó composto pelos sensores $\mathbf{i}_u$. Note-se que, por exemplo, para um nó superior sem elevador, o número de auto-rádios detectados a montante pode variar de 0 até 3.

2. A modelação de situações de excepção é muito fácil de ser modelada em state-charts, devido especialmente aos conceitos de transições com origem em contornos de super-estado e do conector história (fig. 3.17). Com RdP-shobi, as situações de excepção não são habitualmente especificadas estruturalmente pois sobrecarregariam imenso o diagrama, reduzindo a legibilidade do mesmo.
3. Em ambos os meta-modelos, existe a noção de hierarquia estrutural, que tem igualmente associada a noção de hierarquia comportamental. Nos state-charts usam-se super-estados e nas RdP-shobi usam-se macronodos (macrolugares ou macrotransições) para conseguir essa hierarquia estrutural. Com state-charts é possível especificar (definir) o comportamento dum super-estado dentro do seu próprio contorno ou num state-chart à parte. Em RdP-shobi, só é possível esta última opção.
4. As RdP-shobi, para estarem completamente definidas, obrigam a que se modele o sistema controlado. Se por um lado se trata dum factor positivo, pois dá uma perspectiva mais global do sistema, também implica um menor grau de liberdade em relação ao controlo. Nesse sentido, os state-charts podem ser considerados menos rígidos do que as RdP-shobi.
5. As RdP-shobi são mais complicadas de ler do que os state-charts pois apresentam mais elementos de modelação; além do fluxo de controlo aquelas apresentam também os recursos do sistema controlado em uso.
6. A análise de propriedades relativas ao sistema modelado com RdP-shobi passa pela transformação desta para uma RdP-SI, fazendo-se depois as validações neste meta-modelo. Com state-charts, a prova formal de propriedades pode fazer-se com recurso, por exemplo, a Real-Time Logic [Barroca, 1992] [Armstrong e Barroca, 1996].

O uso de redes de Petri, em detrimento de state-charts, para especificar o comportamento de objectos tem sido sugerido por vários autores [Giese et al., 1999], entre os quais o próprio Harel, "pai" dos state-charts [Harel, 1999].

7.2.13 Especificação Oblog

Com base nos diversos diagramas construídos para o SCLH, foi possível obter uma especificação escrita em código Oblog [Team BP-UM, 1999a, pág. 77–116], seguindo algumas das regras de transformação sugeridas no cap. 6. Dado o elevado número de páginas que ocuparia, esse código não se reproduz neste trabalho.

7.2.14 Comentários

Este exemplo, bem mais complexo que o SSI, permitiu abordar diversas questões de modelação. Além doutros tópicos, com o SCLH, foi possível mostrar a forma como um caso de uso pode ser decomposto em outros casos de uso; a passagem dos casos de uso para objectos; a necessidade em descrever pormenorizadamente, durante o desenvolvimento dum sistema embebido, o sistema controlado; a construção dum diagrama de classes como meio para organizar os vários tipos de componentes do sistema; e o modo como os state-charts podem ser usados para formalizar os requisitos do sistema, descritos sob a forma de regras e cenários.

Relativamente aos cenários de funcionamento, foi utilizada para a sua descrição uma notação facilmente entendida pelo cliente, em detrimento dos diagramas de interacção. Este exemplo mostra que o projectista não se deve limitar ao uso dos diagramas disponíveis em UML, mas que deve, antes, procurar documentar os projectos com notações que possam auxiliar os seus possíveis leitores.

Este exemplo mostrou igualmente que o recurso a outras técnicas, não previstas inicialmente na metodologia, deve ser incentivado, caso tal se revele útil. Foi o que se verificou, neste caso, relativamente à inclusão da simulação. Para sistemas de controlo, parece pertinente o estudo do impacto das estratégias de controlo na qualidade final do sistema, daí que, segundo o autor, deva equacionar-se a inclusão dessa actividade na metodologia MIDAS.

7.3 Resumo final

Mostrou-se, neste capítulo, a utilização real da metodologia MIDAS a casos práticos, evidenciando algumas questões de modelação que se introduziram em capítulos anteriores. Como exemplos, forma considerados o Sistema de Supervisão de Iluminação (SSI) e o Sistema de Controlo das Linhas HIDRO (SCLH), o que permitiu validar a utilização da metodologia.

A aplicação da metodologia MIDAS a casos práticos e reais, além de se revelar um dado extremamente importante para validar aquela, permitiu simultaneamente equacionar formas de a melhorar. De facto, a definição duma metodologia para desenvolvimento de sistemas, só parece fazer sentido se for aplicada em desenvolvimentos de sistemas reais. O mero exercício mental de criar conceptualmente uma metodologia, apesar de válido e útil, fica, no entender do autor, incompleto se não for acompanhado da demonstração da sua aplicabilidade a casos reais. Daí que as aplicações tenham sido e deverão continuar a sê-lo, no futuro, a grande "mola impulsionadora" para a conceptualização da metodologia MIDAS.

Capítulo 8

Conclusões

Não acabar é não fazer.

Só não se acaba o que nunca se começa.

8.1 Trabalho realizado

O contributo principal desta tese consistiu na definição e validação da metodologia MIDAS. Esta metodologia pretende auxiliar o projectista no desenvolvimento de sistemas embebidos, com destaque para os sistemas complexos a implementar em plataformas mistas (com hardware e software), pelo que pode enquadrar-se MIDAS na área do co-projecto de hardware-software. De entre as várias características que a metodologia MIDAS apresenta, realçam-se as seguintes:

- Modelação orientada ao objecto.
- Especificação unificada, gráfica e multi-vista.
- Abordagem operacional.

Um dos objectivos da tese era mostrar que a abordagem orientada ao objecto, que tão bons resultados tem conseguido na engenharia de software, pode ser igualmente aplicada, com efeitos positivos, na modelação de sistemas embebidos. Importante foi também a constatação que, para lidar com sistemas complexos, é necessário recorrer a especificações unificadas (independentes da implementação final), gráficas (por serem claras e intuitivas, mas, simultaneamente, precisas e rigorosas) e multi-vista (por representarem diferentes vistas do mesmo sistema). Finalmente, pôde concluir-se que a possibilidade em executar as especificações (abordagem operacional) e a adopção dum processo iterativo e incremental são dois factores cruciais para o desenvolvimento de sistemas complexos e, em particular, de sistemas embebidos complexos.

Foi indicado um meta-modelo multi-vista adequado para sistemas embebidos e foram seleccionadas as respectivas notações, que se baseiam totalmente em UML. Indicou-se, igualmente, o modelo de processo e, para cada uma das tarefas identificadas, foi fornecido o respectivo método, composto por algumas recomendações que devem/podem ser seguidas para a persecução da dita tarefa.

Com o intuito de validar e demonstrar a aplicabilidade da metodologia MIDAS, foram descritos, sob o ponto de vista técnico e metodológico, 2 projectos de desenvolvimento de exemplos reais (um sistema de supervisão de iluminação e o sistema de controlo das linhas de produção duma fábrica).

Esta tese centrou a sua atenção nas questões associadas à modelação de sistemas embebidos complexos, nomeadamente o levantamento de requisitos, os meta-modelos a adoptar para especificação e os métodos a seguir na fase de análise. Neste contexto, os assuntos ligados às fases de concepção e implementação não foram discutidos dum modo tão aprofundado.

8.2 Trabalho futuro

Nesta secção, pretende fazer-se o levantamento de algumas questões que foram deixadas em aberto e que merecem, segundo o autor, um tratamento futuro, de modo a melhorar a metodologia MIDAS e alargar, se possível, o seu âmbito de aplicação.

8.2.1 Introdução da metodologia MIDAS em instituições

Hoje em dia, é importante notar-se que a maioria das organizações, que desenvolvem sistemas computacionais, não usa qualquer metodologia. Esta ideia é reflectida pela seguinte citação:

> "*From a methodological perspective, it's important to realize that most organizations worldwide still don't follow any mature, repeatable method.*" [Booch, 1996, pág. 106].

O problema é certamente muito mais flagrante na área dos sistemas embebidos, pois tradicionalmente a apetência por abordagens metodológicas (ou, pelo menos, a consciência da sua necessidade) é muito diminuta. Trata-se portanto duma situação que urge colmatar, e que passa pela introdução, em organizações que desenvolvem sistemas embebidos, das metodologias actualmente existentes. No entanto, a forma como estas são introduzidas nas organizações deve ser ponderada, pois há todo um *status quo* que é preciso alterar.

Actualmente, uma metodologia leva alguns anos até atingir a maturidade. Uma metodologia é o resultado de um processo ascendente, já que só assim se podem analisar dificuldades, procurar soluções, escolher meta-modelos e métodos e validar propostas. Nenhuma metodologia pode ser aceite, sem ter sido antes sujeita a experimentação.

Ler um livro sobre uma metodologia para desenvolvimento de sistemas embebidos não é suficiente para que o seu leitor, mesmo sendo engenheiro informático ou electrotécnico, possa ser considerado apto a usá-la. Só a sua utilização em projectos reais permite ao engenheiro adquirir a "bagagem" suficiente para poder aproveitar, com rapidez e eficiência, a metodologia.

Daí que, relativamente à metodologia MIDAS, haja a sensação que a sua introdução em instituições requer um período de formação inicial, um acompanhamento profundo nos primeiros sistemas a desenvolver e um constante apoio técnico que permita a evolução da metodologia e de quem a utiliza.

8.2.2 Envolvimento em mais projectos

Uma metodologia só tem razão de existir se for realmente utilizada para desenvolver sistemas. Por outro lado, só com a aplicação da metodologia a casos reais e concretos é que é possível validar e, concomitantemente, melhorar a metodologia. Daí que, no futuro, haja todo o interesse em "pôr à prova" a metodologia MIDAS no desenvolvimento de mais sistemas embebidos. Tal

só será exequível se algumas organizações adoptarem, para seu próprio benefício, a metodologia MIDAS, tal como sugerido na subsecção anterior.

Neste sentido, decorrem já, em colaboração com a Unidade de Automação Industrial e Electrónica do IDITE-MINHO[1], dois projectos em que será aplicada a metodologia MIDAS.

8.2.3 Fases de concepção e implementação

Esta tese tratou exaustivamente a fase de análise, mas propositadamente dedicou menos espaço às fases de concepção e implementação. No âmbito do grupo a que o autor pertence, duas outras teses de doutoramento, a concluir brevemente, tratam de forma mais completa as várias tarefas associadas a essas 2 fases do processo de desenvolvimento. O conjunto das 3 teses foi idealizado, de modo a que elas se complementassem entre si, razão pela qual os resultados do conjunto podem ser usados para construir uma metodologia completa de desenvolvimento de sistemas embebidos (da análise à implementação, incluindo o teste).

Há a percepção que, ao longo deste trabalho, alguns assuntos foram tratados duma forma pouco aprofundada ou mesmo ignorados, nomeadamente o estudo de viabilidade, a gestão de projecto, o controlo da qualidade, o formato da documentação, a partição ou o teste. Tal não significa que não haja o conhecimento da sua importância, mas apenas que os mesmos não se enquadravam no âmbito da tese realizada. Assim, de futuro, será interessante alargar o âmbito da metodologia, de modo a enquadrar, de forma sustentada e metodológica, algumas das temáticas associadas ao desenvolvimento de sistemas ainda em falta.

8.2.4 UML

A notação UML, usada para a documentação na fase de análise, irá com certeza conhecer, no futuro, novas actualizações. Convém pois estar a par dessas actualizações e incluir aquelas que parecerem relevantes no âmbito dos sistemas embebidos e da metodologia MIDAS.

Nesse sentido, será igualmente importante estar atento à proposta do grupo de trabalho RTAD (*Real-Time Analysis and Design*) da OMG para incorporar directamente, na versão oficial UML, conceitos de tempo-real, nomeadamente no que respeita à sintaxe da linguagem OCL. Outras fontes de inspiração serão também as várias propostas que irão continuar a aparecer na área dos sistemas embebidos e de tempo-real [Herzberg, 1999] [Kabous e Nebel, 1999].

8.2.5 Utilitários de apoio ao projecto

Nesta tese, não houve um forte investimento na construção de utilitários que auxiliem os projectistas no desenvolvimento de sistemas embebidos. É uma forte necessidade a construção dum utilitário CASE que, de alguma forma, dê suporte computacional à metodologia proposta.

Trata-se, no entender do autor, duma tarefa bastante ingrata, pois ao não se pretender impor um modelo de processo rígido, está necessariamente a condicionar-se a ferramenta, ao flexibilizar a sua utilização. Este facto pode ser um grave problema, pois sendo a ferramenta flexível, pode não ser possível garantir que a metodologia é aplicada sem ferir os seus princípios fundamentais.

[1] Informações sobre este instituto de I&D estão disponíveis no seguinte URL: `www.idite-minho.pt`.

8.2.6 Estruturas de classes

Para se equacionar a ampla utilização da metodologia em ambientes de trabalho profissionais, é necessário criar bibliotecas que disponibilizem classes básicas necessárias à maioria das aplicações duma dada área de aplicação. Por exemplo, na área da automação industrial, parece óbvia a necessidade em ter disponíveis, em classes, as seguintes funcionalidades: algoritmos para lógica de controlo, mecanismos para criação de *logs*, gestores de alarmes e interfaces para recolha de informação.

A disponibilidade comercial das referidas bibliotecas de classes torna a metodologia mais utilizável e permite obter ganhos de produtividade que se podem reflectir numa vantagem competitiva, para a equipa de desenvolvimento, em mercados fortemente concorrentes. Por outro lado, a utilização de classes pré-fabricadas, no desenvolvimento dum dado sistema, tem como consequências, entre outros, a redução drástica do tempo de desenvolvimento, o aumento da confiabilidade da aplicação final e a antecipação da sua utilização.

Nesta linha de pensamento, um dos projectos a decorrer em parceria com o IDITE-MINHO, anteriormente referidos, pretende intervir neste tópico, através da criação de módulos directamente utilizáveis em aplicações de supervisão e monitorização nas áreas industriais do têxtil e do vestuário.

8.2.7 Simulação

O recurso à simulação com o ambiente ARENA, no projecto do SCLH (secção 7.2), pôs em evidência a importância dessa técnica, como forma de validar o impacto das estratégias de controlo na produção. Durante esse projecto, não houve inicialmente a noção da necessidade em incluir no processo de desenvolvimento essa tarefa de simulação. Daí que não tenha sido considerada prioritária a inclusão, nesta tese, dum estudo sobre a integração da simulação estatística no processo de desenvolvimento.

Contudo, existe já em curso uma dissertação de mestrado (Engenharia Industrial), cujo tema principal se debruça exactamente sobre essa integração, nomeadamente a forma de automatizar a passagem da especificação em OBLOG para o modelo ARENA, permitindo assim a simulação do sistema com base na respectiva especificação.

Parte III

Apêndices

Apêndice A

Supervisão de iluminação: código do protótipo

Neste apêndice, apresenta-se o código em linguagem JAVA relativo ao protótipo construído com o propósito de validar os requisitos do utilizador captados durante a fase de análise do sistema em causa.

```
/*
 *  <applet code="SSI" width=680 height=310>
 *  </applet>
 *
 */

import java.io.*;
import java.util.*;
import java.awt.*;
import java.applet.*;

// ---------------------------------------------------------
interface Constantes {
  int NUM_LUZ=10;
  int NUM_FACHADAS=5;
  int RELÓGIO=0;
  int FOTO=1;
  int FOTORELÓGIO=2;
  int MANUAL=3;
}

// ---------------------------------------------------------
public class SSI extends Applet implements Constantes {

  // -- Objectos do Sistema <<entity >> --
    ProgramaLigação programaLigação;
    InfoPontosLuz infoPontosLuz;
    InfoHistorial infoHistorial;

  // -- Objectos do Sistema <<function>> --
    Monitorizar monitorizar;
    GerarHistorial gerarHistorial;
    Relógio relógio;
```

```
// -- Objectos do Sistema <<interface>>  [utilizador] --
  Programar programar;
  Dias dias;
  RegistarAlterações registarAlterações;
  MostrarAlarmes mostrarAlarmes;
  Historial historial;
  Label tempo;   // área para mostrar hora e dia

// -- Objectos do Sistema <<interface>>  [sensores e actuadores] --
  FotoCélula fotoCélula;                 // área para foto-célula
  ContactoDisjuntor contactoDisjuntor;   // área para estado contacto
  AparelhoMedida aparelhoMedida;

// -- Objectos de menus --
  CardLayout cl;
  Panel inp;

public void init() {

  setLayout(new BorderLayout());

  // - Botões para escolher operações -
  Panel botões = new Panel();
  botões.setLayout(new FlowLayout(FlowLayout.CENTER));
  Choice opc1 = new Choice();
  opc1.addItem("Relógio");
  opc1.addItem("Foto-Célula");
  opc1.addItem("Foto-Célula+Relógio");
  opc1.addItem("Manual");
  botões.add(new Button("Ligações"));
  botões.add(new Button("Dias"));
  botões.add(new Button("Pontos de Luz"));
  botões.add(new Button("Alarmes"));
  botões.add(new Button("Relatórios"));
  botões.add(new Label("Modo ",Label.RIGHT));
  botões.add(opc1);

  // -- Objectos do Sistema <<interface>>  [sensores e actuadores] --
  Panel sensores = new Panel();
  sensores.setLayout(new FlowLayout(FlowLayout.CENTER));
  fotoCélula = new FotoCélula("Foto-Célula");
  contactoDisjuntor = new ContactoDisjuntor("Disjuntor");
  tempo = new Label("",Label.CENTER);
  tempo.setFont(new Font("Courier", Font.PLAIN, 14));
  aparelhoMedida = new AparelhoMedida();
  sensores.add(tempo);
  sensores.add(fotoCélula);
  sensores.add(contactoDisjuntor);
  sensores.add(aparelhoMedida);

  // -- Objectos do Sistema <<entity >> --
  programaLigação = new ProgramaLigação();
  infoPontosLuz = new InfoPontosLuz();
  infoHistorial = new InfoHistorial();

  historial = new Historial();
  historial.setEditable(false);
  historial.setFont(new Font("Courier", Font.PLAIN, 12));
```

```
  historial.setBackground(Color.black);
  historial.setForeground(Color.white);

  // -- Objectos do Sistema <<function>> --
  monitorizar = new Monitorizar(programaLigação,infoPontosLuz,historial,fotoCélula,
      contactoDisjuntor);
  gerarHistorial = new GerarHistorial(historial,infoHistorial,contactoDisjuntor,
      aparelhoMedida);
  Relógio relógio = new Relógio(monitorizar,gerarHistorial,tempo);

  // -- Área para menus volantes --
  inp = new Panel();
  cl = new CardLayout();
  inp.setLayout(cl);

  // -- Objectos do Sistema <<interface>>  [utilizador] --
  programar = new Programar(programaLigação,historial);
  dias = new Dias(programaLigação,historial);
  registarAlterações = new RegistarAlterações(infoPontosLuz,historial);
  mostrarAlarmes = new MostrarAlarmes(infoHistorial);
  inp.add("MenuProg",programar);
  inp.add("MenuDias",dias);
  inp.add("MenuAltera",registarAlterações);
  inp.add("MenuAlarm",mostrarAlarmes);
  inp.add("MenuRelat",new Label("Gerado relatório com valores medidos a cada hora",
      Label.CENTER));

  // -- Localização das várias áreas --
  add("North",botões);
  add("West",inp);
  add("Center",historial);
  add("South",sensores);
  cl.show(inp,"MenuProg");

  // -- Informações iniciais --
  historial.boasVindas();

  // -- Inicio das threads --
  relógio.arranca();
  monitorizar.arranca();
  //showStatus("Braga, Portugal");
}

public boolean action(Event e, Object o) {

  if ("Ligações".equals(o)) {
    cl.show(inp,"MenuProg");
  }
  else if ("Pontos de Luz".equals(o)) {
    cl.show(inp,"MenuAltera");
  }
  else if ("Alarmes".equals(o)) {
    cl.show(inp,"MenuAlarm");
  }
  else if ("Relatórios".equals(o)) {
    cl.show(inp,"MenuRelat");
    infoHistorial.grava(infoHistorial.medidas);
    historial.limpa();
```

```
    }
    else if ("Dias".equals(o)) {
      cl.show(inp,"MenuDias");
    }
    else if ("Relógio".equals(o) || "Foto-Célula".equals(o) ||
             "Foto-Célula+Relógio".equals(o) || "Manual".equals(o)) {
      programaLigação.setModo(""+o);
      historial.escreve("Modo: "+o,true,true);
    }
    return (true);
  }
}

/*************************************************************************
 *  Classes para objectos <<entity>>
 */

// -----------------------------------------------------
class ProgramaLigação implements Constantes {  // <<entity>>

  // para modo Relógio indica as horas a que ligam os pontos de luz
  boolean horaActiva[][];
  boolean modoRelógio[][][][];
  boolean modoFotocélula[][];
  boolean modoManual[][];
  boolean diaEspecial[][];
  // o modo Relógio+Foto-Célula é dado pela configuração modoRelógio
  int modo = RELÓGIO;

  // Condições que permitem ao objecto monitorizar determinar se as programações
  // de ligação (modos Foto-Célula e Manual) foram alteradas.
  boolean alterouProgramaçãoFotoC = false;
  boolean alterouProgramaçãoManual = false;

  ProgramaLigação() {
    horaActiva = new boolean[2][24];
    // o 1o. índice de modoRelógio indica se o dia é normal=0 ou especial=1
    modoRelógio = new boolean[2][24][NUM_FACHADAS][NUM_LUZ];
    modoFotocélula = new boolean[NUM_FACHADAS][NUM_LUZ];
    modoManual = new boolean[NUM_FACHADAS][NUM_LUZ];
    // diaEspecial: mes 0 a 11, dia de 1 a 31 (índice 0 ignorado)
    diaEspecial = new boolean[12][32];
    // ler programação de ficheiro
  }

  public void setModo(String s) {
    if ("Relógio".equals(s)) {
      modo=RELÓGIO;
    }
    else if ("Foto-Célula".equals(s)) {
      modo=FOTO;
    }
    else if ("Foto-Célula+Relógio".equals(s)) {
      modo=FOTORELÓGIO;
    }
    else if ("Manual".equals(s)) {
      modo=MANUAL;
    }
```

```
  }

  public void limpaDiasEspeciais(int mes) {
    int d;
    for (d=1; d<=31; d++) {
      diaEspecial[mes][d] = false;
    }
  }

  public void limpaModo(boolean modo[][]) {
    int f,i;
    for (f=0; f<NUM_FACHADAS; f++) {
      for (i=0; i<NUM_LUZ; i++) {
        modo[f][i] = false;
      }
    }
  }

  public void limpaModo(int idx, int hora) {
    int f,i;
    for (f=0; f<NUM_FACHADAS; f++) {
      for (i=0; i<NUM_LUZ; i++) {
        modoRelógio[idx][hora][f][i] = false;
      }
    }
  }
}

// ----------------------------------------------------------
class InfoPontosLuz implements Constantes {  // <<entity>>

  int substits[][];
  int potencia[][];  int horasfun[][];
  String fabricante[][];

  InfoPontosLuz() {
    int i,j;
    //try {
      //InputStream f1 = new FileInputStream("./luzes.txt");
      substits = new int[NUM_FACHADAS][NUM_LUZ];
      potencia = new int[NUM_FACHADAS][NUM_LUZ];
      horasfun = new int[NUM_FACHADAS][NUM_LUZ];
      fabricante = new String[NUM_FACHADAS][NUM_LUZ];
      for (i=0; i<NUM_FACHADAS; i++) {
        for (j=0; j<NUM_LUZ; j++) {
          substits[i][j] = i+j;
          potencia[i][j] = i*j;
          horasfun[i][j] = 0;
          fabricante[i][j] = "Osram";
          //substits[i][j] = f1.read()-48;
          //f1.read(); // le o espaco
          //potencia[i][j] = f1.read()-48;
          //f1.read(); // le o return
        }
      }
    //} catch (IOException ex) {
    //  System.out.println("Erro de leitura em luzes.txt");
    //  return;
```

```
    //}
  }
}

// ------------------------------------------------------
class InfoHistorial {  // <<entity>>

  String medidas="";
  String alarmes="";

  InfoHistorial () {
  }

  public void acrescenta(String s, String linha) {
    int hor = new Date().getHours();
    int min = new Date().getMinutes();
    s+=(hor<10?"0":"")+hor+":"+(min<10?"0":"")+min+" "+linha;
  }

  public void grava(String s) {
    System.out.println(s);
    s="";
  }
}

/*****************************************************************************
 *  Classes para objectos <<entity>> e <<function>>
 */

// ------------------------------------------------------
class Monitorizar implements Runnable,Constantes {  // <<function>>

//«««««««««««««««««««««««««««««««««
// Falta verificar se algum ponto de luz está fundido ou avariado!!
//«««««««««««««««««««««««««««««««««

  public boolean novaHora=false;

  private Thread t;
  ProgramaLigação programaLigação;
  InfoPontosLuz infoPontosLuz;
  Historial historial;
  FotoCélula fotoCélula;
  ContactoDisjuntor contactoDisjuntor;

  Monitorizar(ProgramaLigação programaLigação, InfoPontosLuz infoPontosLuz, Historial historial,
              FotoCélula fotoCélula, ContactoDisjuntor contactoDisjuntor) {
    this.programaLigação   = programaLigação;
    this.infoPontosLuz     = infoPontosLuz;
    this.historial         = historial;
    this.fotoCélula        = fotoCélula;
    this.contactoDisjuntor = contactoDisjuntor;
    t = new Thread(this);
    t.setPriority(Thread.NORM_PRIORITY+1);
  }

  public void run() {
    int últimoModo=programaLigação.modo;
```

```
  boolean últimaFoto=fotoCélula.estadoActivo();
  boolean últimoContacto=contactoDisjuntor.estadoActivo();
  liga();
  while (true) {
    if (últimoModo!=programaLigação.modo) {
      últimoModo=programaLigação.modo;
      liga();
    }
    if (últimaFoto!=fotoCélula.estadoActivo()) {
      últimaFoto=fotoCélula.estadoActivo();
      if (programaLigação.modo==FOTO || programaLigação.modo==FOTORELÓGIO)
        liga();
    }
    if (últimoContacto!=contactoDisjuntor.estadoActivo()) {
      historial.escreve(contactoDisjuntor.mensagem(),true,true);
      últimoContacto=contactoDisjuntor.estadoActivo();
    }
    if (novaHora) {
      novaHora=false;
      if (programaLigação.modo==RELÓGIO || programaLigação.modo==FOTORELÓGIO)
        liga();
    }
    if (programaLigação.alterouProgramaçãoFotoC) {
      programaLigação.alterouProgramaçãoFotoC=false;
      if (programaLigação.modo==FOTO)
        liga();
    }
    if (programaLigação.alterouProgramaçãoManual) {
      programaLigação.alterouProgramaçãoManual=false;
      if (programaLigação.modo==MANUAL)
        liga();
    }
    try {
      t.sleep(3000); // espera 3 seg
    } catch(InterruptedException ex) {;}
  }
}

public void liga() {
  int f,i;
  int horaActual = new Date().getHours();
  int minActual  = new Date().getMinutes();
  int mêsActual  = new Date().getMonth();
  int diaActual  = new Date().getDate();
  int idx = (programaLigação.diaEspecial[mêsActual][diaActual] ? 1 : 0);

  switch (programaLigação.modo) {
    // ==========================
    case RELÓGIO:
      if (programaLigação.horaActiva[idx][horaActual]) {
        historial.escreve("Ligação em modo Relógio",true,true);
        for (f=0; f<NUM_FACHADAS; f++) {
          for (i=0; i<NUM_LUZ; i++) {
            if (programaLigação.modoRelógio[idx][horaActual][f][i]) {
              historial.escreve("#",false,false);
              infoPontosLuz.horasfun[f][i]++;
            }
            else {
```

```
          historial.escreve(".",false,false);
        }
      }
      historial.escreve("",false,true);
    }
  }
  else {
    // apagar todos pontos de luz
    historial.escreve("Hora não programada para ligar",true,true);
  }
  break;
// ==========================
case FOTORELÓGIO:
  if (programaLigação.horaActiva[idx][horaActual] &&
      fotoCélula.estadoActivo()) {
    historial.escreve("Ligação em modo Foto-Célula+Relógio",true,true);
    for (f=0; f<NUM_FACHADAS; f++) {
      for (i=0; i<NUM_LUZ; i++) {
        if (programaLigação.modoRelógio[idx][horaActual][f][i]) {
          historial.escreve("#",false,false);
          infoPontosLuz.horasfun[f][i]++;
        }
        else {
          historial.escreve(".",false,false);
        }
      }
      historial.escreve("",false,true);
    }
  }
  else {
    // apagar todos pontos de luz
    historial.escreve("Hora não programada para ligar ou foto-célula não activa",true,true);
  }
  break;
// ==========================
case FOTO:
  if (fotoCélula.estadoActivo()) {
    historial.escreve("Ligação em modo Foto-Célula",true,true);
    for (f=0; f<NUM_FACHADAS; f++) {
      for (i=0; i<NUM_LUZ; i++) {
        if (programaLigação.modoFotocélula[f][i]) {
          historial.escreve("#",false,false);
          infoPontosLuz.horasfun[f][i]++;
        }
        else {
          historial.escreve(".",false,false);
        }
      }
      historial.escreve("",false,true);
    }
  }
  else {
    // apagar todos pontos de luz
    historial.escreve("Foto-célula não activa",true,true);
  }
  break;
// ==========================
case MANUAL:
```

```
        historial.escreve("Ligação em modo Manual",true,true);
        for (f=0; f<NUM_FACHADAS; f++) {
          for (i=0; i<NUM_LUZ; i++) {
            if (programaLigação.modoManual[f][i]) {
              historial.escreve("#",false,false);
              infoPontosLuz.horasfun[f][i]++;
            }
            else {
              historial.escreve(".",false,false);
            }
          }
          historial.escreve("",false,true);
        }
      break;
      // ==========================
    }
  }

  public void arranca() {
    t.start();
  }
}

// ---------------------------------------------------------
class GerarHistorial {  // <<function>>

  Historial historial;
  InfoHistorial infoHistorial;
  AparelhoMedida aparelhoMedida;
  ContactoDisjuntor contactoDisjuntor;

  GerarHistorial (Historial historial, InfoHistorial infoHistorial,
                  ContactoDisjuntor contactoDisjuntor, AparelhoMedida aparelhoMedida) {
    this.historial         = historial;
    this.infoHistorial     = infoHistorial;
    this.contactoDisjuntor = contactoDisjuntor;
    this.aparelhoMedida    = aparelhoMedida;
  }

  public void regista() {
    String msg="U="+aparelhoMedida.lêVoltagem()+
               " I="+aparelhoMedida.lêCorrente()+
               " P="+aparelhoMedida.lêPotência()+
               " "+contactoDisjuntor.mensagem()+" ";
    historial.escreve(msg,true,false);
    infoHistorial.acrescenta(infoHistorial.medidas,msg);
  }
}

// ---------------------------------------------------------
class Relógio implements Runnable {  // <<timer>>

  private Thread t;
  Monitorizar monitorizar;
  GerarHistorial gerarHistorial;
  Label tempo;

  Relógio(Monitorizar monitorizar, GerarHistorial gerarHistorial, Label tempo) {
```

```
    this.monitorizar    = monitorizar;
    this.gerarHistorial = gerarHistorial;
    this.tempo          = tempo;
    t = new Thread(this);
    t.setPriority(Thread.NORM_PRIORITY-1);
    mostraRelógio();
 }

  public void run() {
    while (true) {
      if (new Date().getSeconds()==0) {
        mostraRelógio();  // actualiza relógio a cada minuto
        if (new Date().getMinutes()==0) {  // tenta ligar a cada hora
          monitorizar.novaHora=true;
          try {
            t.sleep(7000); // espera 7 seg
          } catch(InterruptedException ex) {;}
          gerarHistorial.regista();
        }
      }
    }
  }

  public void mostraRelógio() {
   tempo.setText(data());
  }

  String data() {
   int ano = new Date().getYear()+1900;
   Mes mes = new Mes(new Date().getMonth());
   int dia = new Date().getDate();
   int hor = new Date().getHours();
   int min = new Date().getMinutes();
   return(
     "["+(dia<10?"0":"")+dia+"/"+
     new String(mes.letras3())+"/"+
     ano+" - "+
     (hor<10?"0":"")+hor+":"+
     (min<10?"0":"")+min+"]");
  }

  public void arranca() {
    t.start();
  }
}

/*************************************************************************
 *  Classes para objectos  de interface com o utilizador
 */

// ----------------------------------------------------
class Programar extends Panel implements Constantes {  // <<interface>>

  ProgramaLigação programaLigação;
  Historial historial;
  Panel inp;
  CardLayout cl;
  TextField tf;
```

```
Checkbox act;
java.awt.List luzesManual,luzesFotoC,luzesRelógio;

static final int REL_NORMAL=0;   // igual ao índice do array modoRelógio
static final int REL_ESPECIAL=1; // igual ao índice do array modoRelógio
static final int FOTOC=2;
static final int MANUAL=3;
int contexto=REL_NORMAL;

int hora=0;

Programar(ProgramaLigação programaLigação, Historial historial) {
  int i,j;
  this.programaLigação = programaLigação;
  this.historial       = historial;
  setLayout(new BorderLayout());

  Panel zona1 = new Panel();
  zona1.setLayout(new FlowLayout(FlowLayout.CENTER));
  Choice mod = new Choice();
  mod.addItem("Relógio Normal");
  mod.addItem("Relógio Especial");
  mod.addItem("Foto-Célula");
  mod.addItem("Manual");
  zona1.add(new Label("Programação para ",Label.RIGHT));
  zona1.add(mod);

  // - Área para menus volantes -
  inp = new Panel();
  cl = new CardLayout();
  inp.setLayout(cl);

  Panel menuManual = new Panel();
  menuManual.setLayout(new BorderLayout());
  luzesManual = new java.awt.List(NUM_LUZ*NUM_FACHADAS,true);
  for (i=0; i<NUM_FACHADAS; i++) {
    for (j=0; j<NUM_LUZ; j++) {
      luzesManual.addItem(i+","+j);
      if (programaLigação.modoManual[i][j])
        luzesManual.select(i* NUM_FACHADAS+j);
      else
       luzesManual.deselect(i* NUM_FACHADAS+j);
    }
  }
  menuManual.add("Center",luzesManual);

  Panel menuFotoC = new Panel();
  menuFotoC.setLayout(new BorderLayout());
  luzesFotoC = new java.awt.List(NUM_LUZ*NUM_FACHADAS,true);
  for (i=0; i<NUM_FACHADAS; i++) {
    for (j=0; j<NUM_LUZ; j++) {
      luzesFotoC.addItem(i+","+j);
      if (programaLigação.modoFotocélula[i][j])
        luzesFotoC.select(i* NUM_FACHADAS+j);
      else
       luzesFotoC.deselect(i* NUM_FACHADAS+j);
    }
  }
```

```
    menuFotoC.add("Center",luzesFotoC);

    Panel menuRelógio = new Panel();
    menuRelógio.setLayout(new BorderLayout());
    luzesRelógio = new java.awt.List(NUM_LUZ*NUM_FACHADAS,true);
    for (i=0; i<NUM_FACHADAS; i++) {
      for (j=0; j<NUM_LUZ; j++) {
        luzesRelógio.addItem(i+","+j);
        if (programaLigação.modoRelógio[0][0][i][j])
          luzesRelógio.select(i*NUM_FACHADAS+j);
        else
         luzesRelógio.deselect(i*NUM_FACHADAS+j);
      }
    }
    Panel zonaHora = new Panel();
    zonaHora.setLayout(new FlowLayout(FlowLayout.CENTER));
    zonaHora.add(new Button("-"));
    zonaHora.add(new Button("+"));
    zonaHora.add(new Label("Hora: ",Label.RIGHT));
    tf = new TextField("0",2);
    tf.setEditable(false);
    zonaHora.add(tf);
    act = new Checkbox("Activa");
    act.setState(programaLigação.horaActiva[0][0]);
    zonaHora.add(act);
    menuRelógio.add("Center",luzesRelógio);
    menuRelógio.add("South",zonaHora);

    inp.add("MenuManual",menuManual);
    inp.add("MenuFotoC",menuFotoC);
    inp.add("MenuRelógio",menuRelógio);

    Panel altera = new Panel();
    altera.setLayout(new FlowLayout(FlowLayout.CENTER));
    altera.add(new Button("Altera"));

    add("North",zona1);
    add("Center",inp);
    add("South",altera);
    cl.show(inp,"MenuRelógio");
  }

  void refresca() {
    int i,j;
    act.setState(programaLigação.horaActiva[contexto][hora]);
    tf.setText(""+hora);
    for (i=0; i<NUM_FACHADAS; i++) {
      for (j=0; j<NUM_LUZ; j++) {
        if (programaLigação.modoRelógio[contexto][hora][i][j])
          luzesRelógio.select(i*NUM_FACHADAS+j);
        else
         luzesRelógio.deselect(i*NUM_FACHADAS+j);
      }
    }
  }

  public boolean action(Event e, Object o) {
    int i;
```

```
int fax=0;
int pl=0;
int virg;
if ("+".equals(o)) {
  hora=(hora+1)%24;
  refresca();
}
else if ("-".equals(o)) {
  if (hora==0)
    hora=23;
  else
    hora--;
  refresca();
}
else if ("Relógio Normal".equals(o)) {
  cl.show(inp,"MenuRelógio");
  contexto = REL_NORMAL;
  refresca();
}
else if ("Relógio Especial".equals(o)) {
  cl.show(inp,"MenuRelógio");
  contexto = REL_ESPECIAL;
  refresca();
}
else if ("Foto-Célula".equals(o)) {
  cl.show(inp,"MenuFotoC");
  contexto = FOTOC;
}
else if ("Manual".equals(o)) {
  cl.show(inp,"MenuManual");
  contexto = MANUAL;
}
else if ("Altera".equals(o)) {
  switch (contexto) {
    case REL_NORMAL:    // --- Relógio Normal
    case REL_ESPECIAL:  // --- Relógio Especial
      programaLigação.horaActiva[contexto][hora]=act.getState();
      programaLigação.limpaModo(contexto,hora);
      for (i=0; i<luzesRelógio.getSelectedItems().length; i++) {
        virg = luzesRelógio.getSelectedItems()[i].indexOf(',');
        Integer v = new Integer(0);
        fax = v.parseInt(luzesRelógio.getSelectedItems()[i].substring(0,virg));
        pl =  v.parseInt(luzesRelógio.getSelectedItems()[i].substring(virg+1));
        programaLigação.modoRelógio[contexto][hora][fax][pl]=true;
      }
      if (contexto==REL_NORMAL)
        historial.escreve("Programação Relógio Normal alterada",true,true);
      else
        historial.escreve("Programação Relógio Especial alterada",true,true);
      break;
    case FOTOC:  // --- Foto-Célula
      programaLigação.limpaModo(programaLigação.modoFotocélula);
      for (i=0; i<luzesFotoC.getSelectedItems().length; i++) {
        virg = luzesFotoC.getSelectedItems()[i].indexOf(',');
        Integer v = new Integer(0);
        fax = v.parseInt(luzesFotoC.getSelectedItems()[i].substring(0,virg));
        pl =  v.parseInt(luzesFotoC.getSelectedItems()[i].substring(virg+1));
        programaLigação.modoFotocélula[fax][pl]=true;
```

```
          }
          historial.escreve("Programação Foto-Célula alterada",true,true);
          programaLigação.alterouProgramaçãoFotoC = true;
          break;
        case MANUAL:  // --- Manual
          programaLigação.limpaModo(programaLigação.modoManual);
          for (i=0; i<luzesManual.getSelectedItems().length; i++) {
            virg = luzesManual.getSelectedItems()[i].indexOf(',');
            Integer v = new Integer(0);
            fax = v.parseInt(luzesManual.getSelectedItems()[i].substring(0,virg));
            pl =  v.parseInt(luzesManual.getSelectedItems()[i].substring(virg+1));
            programaLigação.modoManual[fax][pl]=true;
          }
          historial.escreve("Programação Manual alterada",true,true);
          programaLigação.alterouProgramaçãoManual = true;
          break;
      }
    }
    return (true);
  }
}

// ---------------------------------------------------------
class Dias extends Panel {  // <<interface>>

  ProgramaLigação programaLigação;
  Historial historial;
  TextField tf; // área para mês (dias)
  Mes m;
  // 3 listas diferentes conforme o numero de dias dos meses
  java.awt.List lista29;
  java.awt.List lista30;
  java.awt.List lista31;
  Panel listaDias;
  CardLayout cl;

  Dias(ProgramaLigação programaLigação, Historial historial) {
    int i;

    setLayout(new BorderLayout());
    this.programaLigação = programaLigação;
    this.historial       = historial;
    m = new Mes(0);

    Panel zonaMes = new Panel();
    zonaMes.setLayout(new FlowLayout(FlowLayout.CENTER));
    zonaMes.add(new Button("Anterior"));
    zonaMes.add(new Button("Seguinte"));
    zonaMes.add(new Label("Mês: ",Label.RIGHT));
    tf = new TextField("Jan",3);
    tf.setEditable(false);
    zonaMes.add(tf);

    listaDias = new Panel();
    listaDias.resize(100,100);
    cl = new CardLayout();
    listaDias.setLayout(cl);
    lista29 = new java.awt.List(29,true);
```

```
  lista30 = new java.awt.List(30,true);
  lista31 = new java.awt.List(31,true);
  for (i=1; i<=29; i++)
    lista29.addItem(""+i);
  for (i=1; i<=30; i++)
    lista30.addItem(""+i);
  for (i=1; i<=31; i++)
    lista31.addItem(""+i);
  listaDias.add("31",lista31);  // lista inicial para Janeiro
  listaDias.add("29",lista29);
  listaDias.add("30",lista30);

  Panel altera = new Panel();
  altera.setLayout(new FlowLayout(FlowLayout.CENTER));
  altera.add(new Button("Altera"));

  add("North",zonaMes);
  add("Center",listaDias);
  add("South",altera);
}

void refresca() {
  int i;
  tf.setText(m.letras3());
  // Fev
  if (m.mes==1) {
    cl.show(listaDias,"29");
    for (i=1; i<=29; i++)
      if (programaLigação.diaEspecial[m.mes][i])
        lista29.select(i-1); //dia 1 está no índice 0
      else
        lista29.deselect(i-1);
    }
  // Abr,Jun,Set,Nov
  else if  (m.mes==3 || m.mes==5 || m.mes==8 || m.mes==10) {
    cl.show(listaDias,"30");
    for (i=1; i<=30; i++)
      if (programaLigação.diaEspecial[m.mes][i])
        lista30.select(i-1); //dia 1 está no índice 0
      else
        lista30.deselect(i-1);
  }
  // Jan,Mar,Mai,Jul,Ago,Out,Dez
  else {
    cl.show(listaDias,"31");
    for (i=1; i<=31; i++)
      if (programaLigação.diaEspecial[m.mes][i])
        lista31.select(i-1); //dia 1 está no índice 0
      else
        lista31.deselect(i-1);
  }
}

public boolean action(Event e, Object o) {
  int i;
  if ("Seguinte".equals(o)) {
    m.mes=(m.mes+1)%12;
    refresca();
```

```
    }
    else if ("Anterior".equals(o)) {
      if (m.mes==0)
        m.mes=11;
      else
        m.mes--;
      refresca();
    }
    else if ("Altera".equals(o)) {
      programaLigação.limpaDiasEspeciais(m.mes);
      if (m.mes==1) { // Fev
        for (i=0; i<lista29.getSelectedIndexes().length; i++) {
          programaLigação.diaEspecial[m.mes][1+lista29.getSelectedIndexes()[i]]=true;
        }
      }
      else if  (m.mes==3 || m.mes==5 || m.mes==8 || m.mes==10) { // Abr,Jun,Set,Nov
        for (i=0; i<lista30.getSelectedIndexes().length; i++) {
          programaLigação.diaEspecial[m.mes][1+lista30.getSelectedIndexes()[i]]=true;
        }
      }
      else { // Jan,Mar,Mai,Jul,Ago,Out,Dez
        for (i=0; i<lista31.getSelectedIndexes().length; i++) {
          programaLigação.diaEspecial[m.mes][1+lista31.getSelectedIndexes()[i]]=true;
        }
      }
      historial.escreve("Dias Normais/Especiais alterados: "+m.letras3(),true,true);
    }
    return (true);
  }
}

// ---------------------------------------------------------
class RegistarAlterações extends Panel implements Constantes {  // <<interface>>

  InfoPontosLuz infoPontosLuz;
  Historial historial;
  TextField tf0;  // área para num. fachada
  TextField tf1;  // área para num. ponto luz
  TextField tf2;  // área para potência
  TextField tf3;  // área para subsitituições
  TextField tf4;  // área para horas funcionamento
  TextField tf5;  // área para fabricante
  int pl;         // valor do ponto de luz a mostrar
  int fax;        // valor da fachada

  RegistarAlterações(InfoPontosLuz infoPontosLuz, Historial historial) {
    this.infoPontosLuz = infoPontosLuz;
    this.historial     = historial;
    setLayout(new BorderLayout());
    Panel zona1 = new Panel();
    Panel zona2 = new Panel();
    Panel zona3 = new Panel();
    zona1.setLayout(new FlowLayout(FlowLayout.CENTER));
    zona1.add(new Button("Anterior"));
    zona1.add(new Button("Seguinte"));
    zona1.add(new Label("Fachada: "));
    tf0 = new TextField(""+fax);
    tf0.setEditable(false);
```

```
  zona1.add(tf0);
  zona2.setLayout(new GridLayout(5,2,0,0));
  tf1 = new TextField(""+pl);
  tf1.setEditable(false);
  tf2 = new TextField(""+infoPontosLuz.potencia[fax][pl]);
  tf3 = new TextField(""+infoPontosLuz.substits[fax][pl]);
  tf4 = new TextField(""+infoPontosLuz.horasfun[fax][pl]);
  tf5 = new TextField(infoPontosLuz.fabricante[fax][pl]);
  zona2.add(new Label("Ponto Luz:",Label.RIGHT));
  zona2.add(tf1);
  zona2.add(new Label("Potência:",Label.RIGHT));
  zona2.add(tf2);
  zona2.add(new Label("Substituições:",Label.RIGHT));
  zona2.add(tf3);
  zona2.add(new Label("Funcionamento (h):",Label.RIGHT));
  zona2.add(tf4);
  zona2.add(new Label("Fabricante:",Label.RIGHT));
  zona2.add(tf5);
  zona3.setLayout(new FlowLayout(FlowLayout.CENTER));
  zona3.add(new Button("<<"));
  zona3.add(new Button("<"));
  zona3.add(new Button("Altera"));
  zona3.add(new Button(">"));
  zona3.add(new Button(">>"));
  add("North",zona1);
  add("Center",zona2);
  add("South",zona3);
}

void refresca() {
  tf0.setText(""+fax);
  tf1.setText(""+pl);
  tf2.setText(""+infoPontosLuz.potencia[fax][pl]);
  tf3.setText(""+infoPontosLuz.substits[fax][pl]);
  tf4.setText(""+infoPontosLuz.horasfun[fax][pl]);
}

public boolean action(Event e, Object o) {
  if ("Seguinte".equals(o)) {
    fax=(fax+1)%NUM_FACHADAS;
    refresca();
  }
  else if ("Anterior".equals(o)) {
    if (fax==0)
      fax=NUM_FACHADAS-1;
    else
      fax--;
    refresca();
  }
  else if ("<<".equals(o)) {
    pl=0;
    refresca();
  }
  else if (">>".equals(o)) {
    pl=NUM_LUZ-1;
    refresca();
  }
  else if ("<".equals(o)) {
```

```
      if (pl>0)
        pl--;
      refresca();
    }
    else if (">".equals(o)) {
      if (pl<NUM_LUZ-1)
        pl++;
      refresca();
    }
    else if ("Altera".equals(o)) {
      try {
        Integer v = new Integer(0);
        infoPontosLuz.potencia[fax][pl]=v.parseInt(tf2.getText());
        infoPontosLuz.substits[fax][pl]=v.parseInt(tf3.getText());
        infoPontosLuz.horasfun[fax][pl]=v.parseInt(tf4.getText());
        historial.escreve("Dados alterados: ponto luz ("+fax+","+pl+")",true,true);
      } catch (NumberFormatException ex) {
          historial.escreve("Retifique dados introduzidos",false,true);
      }
    }
    return (true);
  }
}

// ---------------------------------------------------------
class MostrarAlarmes extends Panel {  // <<interface>>

  InfoHistorial infoHistorial;

  MostrarAlarmes (InfoHistorial infoHistorial) {
    this.infoHistorial = infoHistorial;
    setLayout(new FlowLayout(FlowLayout.CENTER));
    add(new Button("Gravar"));
    add(new Label("ALARMES",Label.CENTER));
    add(new Button("Limpar"));
  }

  public boolean action(Event e, Object o) {
    if ("Gravar".equals(o)) {
      infoHistorial.grava(infoHistorial.alarmes);
    }
    else if ("Limpar".equals(o)) {
      infoHistorial.alarmes="";
    }
    return (true);
  }
}

// ---------------------------------------------------------
class Historial extends TextArea {  // <<interface>>

  public void escreve(String s, boolean data, boolean nl) {
    int tamanho=getText().length();
    String d = "";
    if (data) {
      int hor = new Date().getHours();
      int min = new Date().getMinutes();
      d=((hor<10?"0":"")+hor+":"+(min<10?"0":"")+min+" ");
```

```
    }
    insertText(d+s+(nl?"\n":""),tamanho);
  }

  public void limpa() {

    int tamanho=getText().length();
    replaceText("",0,tamanho);
    boasVindas();
  }

  public void boasVindas() {
    escreve("SSI v.1.0 by miguel@di.uminho.pt",false,true);
    escreve("Modo: Relógio",true,true);
  }
}

/**************************************************************************
 *  Classes para objectos sensores e actuadores
 */

// ------------------------------------------------------
class FotoCélula extends Checkbox {  // <<interface>>

  FotoCélula(String t) {
    setLabel(t);
  }

  boolean estadoActivo() {
    return(getState());
  }
}

// ------------------------------------------------------
class ContactoDisjuntor extends Checkbox {  // <<interface>>

  ContactoDisjuntor(String t) {
    setLabel(t);
  }

  public boolean estadoActivo() {
    return(getState());
  }

  public String mensagem() {
    if (estadoActivo())
      return("Disjuntor está disparado");
    else
      return("Disjuntor está normal");
  }

}

// ------------------------------------------------------
class AparelhoMedida extends Panel {  // <<interface>>

  // int voltagem;
  // int corrente;
```

```
  // int potência;
  TextField tfV, tfI, tfP;

  AparelhoMedida() {
    setLayout(new FlowLayout(FlowLayout.CENTER));
    add(new Label("U (V):",Label.RIGHT));
    tfV = new TextField("100",3);
    add(tfV);
    add(new Label("I (A):",Label.RIGHT));
    tfI = new TextField("120",3);
    add(tfI);
    add(new Label("P (W):",Label.RIGHT));
    tfP = new TextField("140",3);
    add(tfP);
  }

  public int lêVoltagem() {
    Integer v = new Integer(0);
    return(v.parseInt(tfV.getText()));
  }

  public int lêCorrente() {
    Integer v = new Integer(0);
    return(v.parseInt(tfI.getText()));
  }

  public int lêPotência () {
    Integer v = new Integer(0);
    return(v.parseInt(tfP.getText()));
  }
}

/*****************************************************************************
 *  Classes utilitárias
 */

// -----------------------------------------------------
class Mes {

  int mes;

  Mes(int mes) {
    this.mes = mes;
  }

  public String letras3() {
    switch(mes) {
      case 0:
        return("Jan");
      case 1:
        return("Fev");
      case 2:
        return("Mar");
      case 3:
        return("Abr");
      case 4:
        return("Mai");
      case 5:
```

```
        return("Jun");
      case 6:
        return("Jul");
      case 7:
        return("Ago");
      case 8:
        return("Set");
      case 9:
        return("Out");
      case 10:
        return("Nov");
      case 11:
        return("Dez");
      default:
        return("???");
    }
  }
}

// -------------------------------------------------------

// <EndOfFile>
```

Apêndice B

Caracterização das linhas HIDRO

Dado que o equipamento de fabrico das linhas HIDRO é complexo, inclui-se, neste apêndice, informação mais elaborada que permite complementar aquela apresentada na subsecção 7.2.7. Com a excepção de ligeiras alterações de pormenor, este apêndice é uma cópia do cap. 3 do documento do projecto [Team BP-UM, 1999a], que resultou dos contributos das diversas pessoas que nele participaram (em que se inclui o autor desta tese).

B.1 Equipamento de produção

Nesta secção, são apresentados os diversos equipamentos de fabrico das linhas HIDRO, que são compostos por sensores, actuadores e outros elementos mecânicos. Relativamente aos sensores e actuadores (objectos do sistema controlado), é utilizada uma notação que permite que alguns desses objectos possam ser parcialmente referidos (ou seja, não serem designados pela sua referência completa). É habitual, por questões de simplicidade e legibilidade, surgirem diagramas em que as referências dos objectos não utilizam todos os parâmetros de localização topológica, omitindo, por exemplo, os parâmetros referentes ao índice do nó.

B.1.1 Linhas de transporte superior

As linhas de transporte superior alimentam os vários postos de trabalho $\mathbf{P}_{n,site} : n \in \{1 \ldots \mathbf{k}_p\} \wedge site \in \{l, r\}$, até os auto-rádios serem processados. A constante $\mathbf{k}_p$ é configurável, sendo o número máximo de postos de uma linha HIDRO igual a $2\mathbf{k}_p$. A fig. B.1 apresenta uma planta genérica de uma linha de transporte superior que poderá ser designada de *nó básico superior*, uma vez que, a partir desta planta, é possível obter praticamente toda a parte superior das linhas HIDRO.

Um nó básico superior é composto pelos seguintes sensores e actuadores, que constituem os componentes do sistema controlado:

Passadeiras de condução $\mathbf{m}_c$: Este actuador corresponde às passadeiras de condução das linhas de transporte superior que deslocam as paletes no sentido crescente do eixo Ox.

Indutivo vertical $\mathbf{i}_u$: Este sensor permite verificar a (in)existência de uma palete à entrada do transfer **t**.

Leitor b: Este sensor corresponde ao leitor de código de barras que permite identificar um

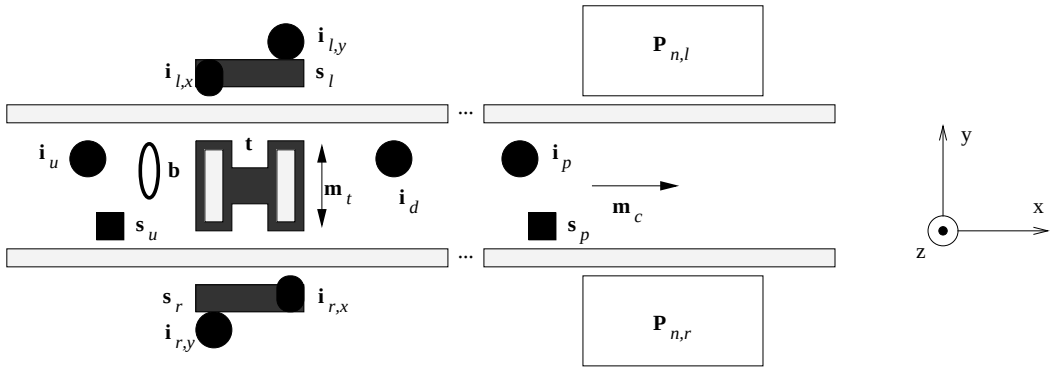


Figura B.1: Planta genérica dum nó básico superior.

auto-rádio que se encontre em cima da palete detectada por $\mathbf{i}_u$.

Stopper $\mathbf{s}_u$: Este actuador permite controlar a entrada de uma palete no transfer **t**.

Passadeiras do transfer $\mathbf{m}_t$: Este actuador (acoplado ao transfer **t**) corresponde às passadeiras que permitem deslocar, ao longo do eixo Oy, a palete que se encontre em cima do transfer **t**, para linhas de transporte superior contíguas ou para elevadores.

Batentes $\mathbf{s}_l$ e $\mathbf{s}_r$: Estes actuadores controlam o deslocamento, ao longo do eixo Oy, da palete que se encontre em cima do transfer **t**, para linhas de transporte superior contíguas ou para elevadores, podendo ser controlados para realizar movimentos ao longo do eixo Oz.

Indutivos horizontais $\mathbf{i}_{l,x}$ e $\mathbf{i}_{r,x}$: Estes sensores (acoplados a $\mathbf{s}_l$ e $\mathbf{s}_r$, respectivamente) detectam a (in)existência de uma palete em cima do transfer **t**, aquando de um deslocamento da palete ao longo do eixo Ox (transfer **t** vazio, se sensores com saída a OFF). Sempre que os sensores $\mathbf{i}_{l,x}$ e $\mathbf{i}_{r,x}$ se encontram acoplados a batentes $\mathbf{s}_l$ e $\mathbf{s}_r$ que estão a ser controlados para realizar movimentos ao longo de Oz, acumulam as funções dos sensores indutivos verticais $\mathbf{i}_{l,y}$ e $\mathbf{i}_{r,y}$.

Indutivos verticais $\mathbf{i}_{l,y}$ e $\mathbf{i}_{r,y}$: Estes sensores (acoplados a $\mathbf{s}_l$ e $\mathbf{s}_r$, respectivamente) detectam a (in)existência de uma palete em cima do transfer **t**, aquando de um deslocamento da palete ao longo do eixo Oy (transfer **t** vazio, se sensores com saída a OFF).

Indutivo vertical $\mathbf{i}_d$: Este sensor permite verificar a (in)existência de uma palete à saída do transfer **t** e, nas linhas de transporte superior com postos, coincide com a última posição do *buffer* dos postos dessa linha (*buffer* cheio, se sensor com saída a ON).

Indutivo vertical $\mathbf{i}_p$: Este sensor, que existe unicamente nas linhas de transporte superior com postos, coincide com a primeira posição do *buffer* dos postos dessa linha (*buffer* não vazio, se sensor com saída a ON).

Stopper $\mathbf{s}_p$: Este actuador, presente apenas nas linhas de transporte superior com postos, controla a saída das paletes do *buffer* dos postos dessa linha.

Na fig. B.1, todos os objectos do sistema controlado, com excepção dos postos, são referenciados com uma notação parcial, em que se omitem os parâmetros *line* (índice da linha) e n (índice do nó). Todavia, o parâmetro n, após instanciado, terá que ser igual em todos os objectos presentes na figura, ou seja, o parâmetro n permite replicar o nó básico superior no eixo Ox, enquanto que parâmetro *line* permite que se replique o nó básico superior no eixo Oy.

Se se replicar o nó básico superior no eixo Oy (instanciando o parâmetro *line* com A, B e C, e

mantendo indefinido o parâmetro n) e se se concatenarem as três instâncias obtidas, é possível construir um *nó composto superior* (fig. B.2) que representa a planta genérica das três linhas de transporte superior.

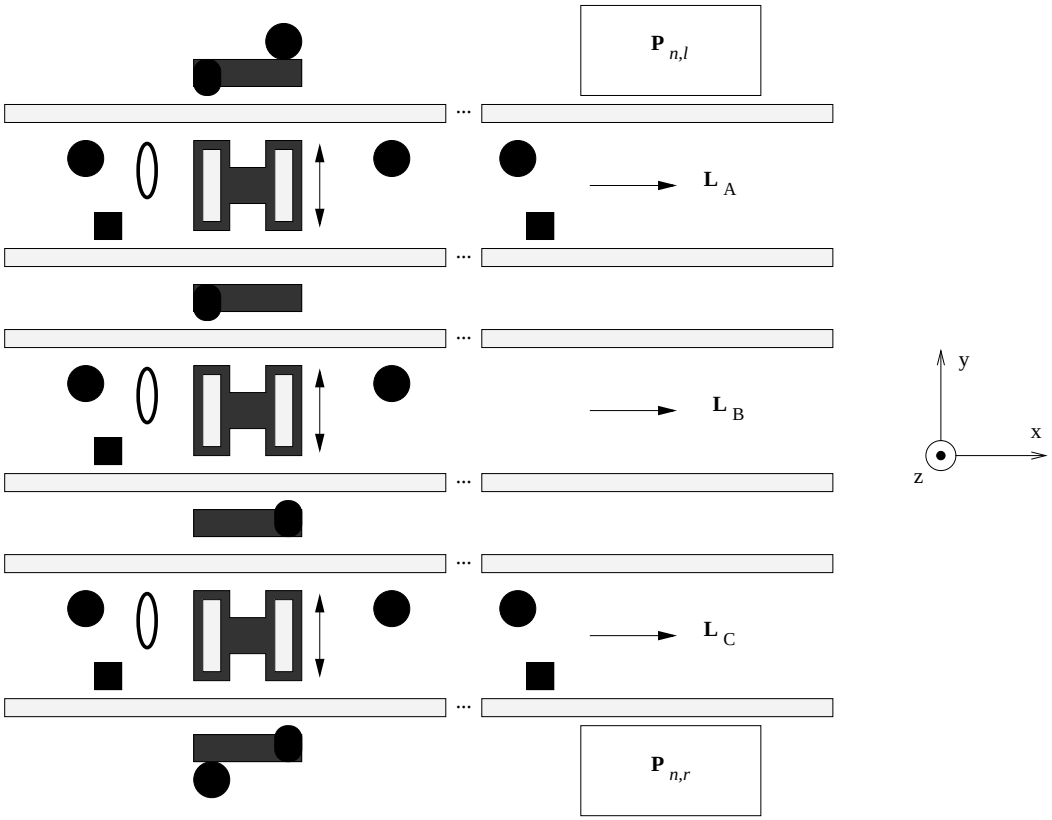


Figura B.2: Planta genérica dum nó superior composto.

Se a fig. B.1 servir como legenda, torna-se possível não utilizar referências para alguns objectos do sistema controlado (recorrendo unicamente aos respectivos símbolos gráficos), porque os parâmetros *site* estão implícitos pela localização topológica dos objectos, caso se garanta que as replicações efectuadas estão perfeitamente explícitas. Por exemplo, na fig. B.2 foi somente necessário utilizar as referências $\mathbf{L}_A$, $\mathbf{L}_B$ e $\mathbf{L}_C$, assumindo-se que os parâmetros *line* de todos os outros objectos, para cada um dos nós básicos que constituem o nó composto superior, estão implicitamente instanciados com **A**, **B** e **C**, respectivamente.

A obtenção do nó composto superior, a partir da replicação e concatenação de nós básicos superiores, exigiu a realização de alguns ajustes, nomeadamente a eliminação de vários objectos:

1. Não são necessários os objectos do acoplamento mecânico $\mathbf{s}_{A,r}\mathbf{i}_{A,r,x}$ (ver secção B.2), uma vez que a função é realizada pelo actuador $\mathbf{s}_{B,l}$. O sensor $\mathbf{i}_{A,r,x}$ não é necessário, uma vez que não é preciso detectar deslocações de paletes, para $\mathbf{L}_A$, no sentido decrescente do eixo Oy, uma vez que não há nenhuma linha à esquerda de $\mathbf{L}_A$.
2. Por razões semelhantes às anteriores, não se torna necessária a existência dos objectos do acoplamento mecânico $\mathbf{s}_{C,l}\mathbf{i}_{C,l,x}$. Contudo, os nós superiores que contenham elevadores junto ao acoplamento mecânico $\mathbf{s}_{C,r}\mathbf{i}_{C,r,x}$, são uma excepção pois exigem a presença do sensor $\mathbf{i}_{C,l,x}$ para detectar deslocações de paletes, dos elevador para $\mathbf{L}_C$, no sentido crescente do eixo Oy (i.e. vindas do elevador).

3. Em $\mathbf{L}_A$ não existe o posto $\mathbf{P}_r$ e em $\mathbf{L}_C$ não existe o posto $\mathbf{P}_l$.
4. Não são necessários $\mathbf{i}_{B,p}$ e $\mathbf{s}_{B,p}$, porque $\mathbf{L}_B$ não possui nenhum dos postos. Verifica-se que quando o acoplamento mecânico $\mathbf{s}_{site}\mathbf{i}_{site,x} \wedge site \in \{l, r\}$ não se encontra nos extremos de condução ao longo do eixo Oy não existe a necessidade de dispor de um sensor indutivo vertical $\mathbf{i}_{site,y}$, uma vez que $\mathbf{s}_{site} \wedge site \in \{l, r\}$ não é controlado para realizar movimentos ao longo de Oz e, como tal, $\mathbf{i}_{site,x}$ acumula as funções de $\mathbf{i}_{site,y}$.

Os ajustes efectuados não levantam problemas aos modelos computacionais das linhas HIDRO, pois não se pretende realizar a síntese do sistema controlado, mas unicamente a síntese de um protótipo do controlador. Estes ajustes vão implicar, após a compilação dos modelos, a inexistência de estados de controlo (no protótipo sintetizado) para os objectos do sistema controlado que entretanto foram eliminados. O mesmo efeito é obtido nas situações em que se mantêm, nas plantas genéricas, objectos do sistema controlado que não são referenciados nos ciclos de controlo a apresentar posteriormente. Todo este processo de síntese é realizado automaticamente, pois a prototipagem é efectuada com geração automática de código.

Com a excepção dos postos, todos os outros objectos do sistema controlado das linhas de transporte podem tomar valores não inteiros para o índice do nó (no formato obrigatório de números compostos fraccionários, com fracção não reduzida), permitindo assim, por exemplo, adicionar nós, sem postos, às linhas HIDRO, sem implicar a atribuição de novos índices aos nós já existentes. Na fig. B.3(a) apresenta-se um exemplo, em que se pretende adicionar, entre os nós $n = 7$ e $n = 8$ das linhas de transporte superior, 3 nós compostos superiores, sem postos, sem sensores $\mathbf{i}_p$ e sem actuadores $\mathbf{s}_p$. Com o esquema de numeração adoptado, todos os nós anteriormente existentes mantêm inalterável o seu índice de nó e os três novos nós passam a possuir os seguintes índices $n = 7\frac{1}{4}$, $n = 7\frac{2}{4}$ e $n = 7\frac{3}{4}$, como a fig. B.3(b) ilustra.

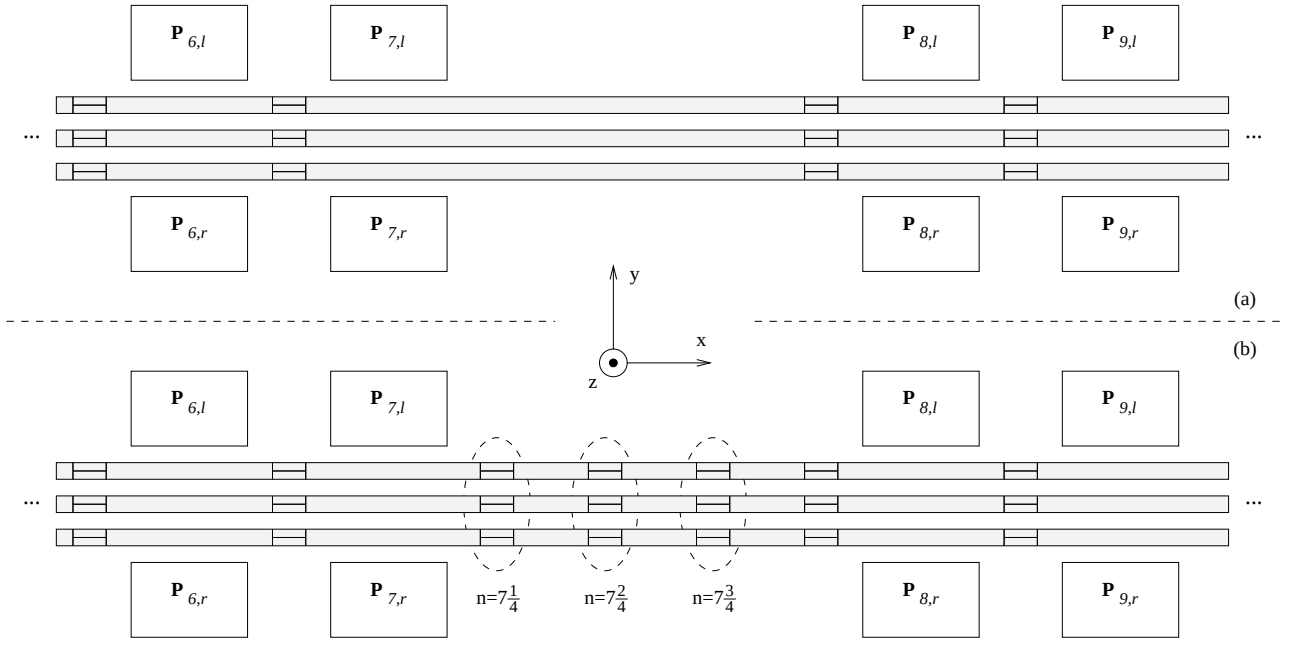


Figura B.3: Adição de nós superiores às linhas HIDRO: (a) Linhas de transporte superior sem nós entre $\mathbf{P}_7$ e $\mathbf{P}_8$; (b) Linhas de transporte superior com três novos nós (sem postos) entre $\mathbf{P}_7$ e $\mathbf{P}_8$.

Este formato de índice proporciona as seguintes vantagens:

1. Evita a alteração dos índices de todos os nós a jusante das alterações efectuadas (e, obviamente, de todos os nós a montante);
2. Garante uma ordem total na sequência dos nós existentes, em cada instante, nas linhas;
3. Fornece informação precisa quanto ao número de nós, sem postos, existentes entre cada dois nós, com índice inteiro. Note-se que o denominador da fracção do índice dos nós não designa o número de nós, sem postos, adicionados entre outros dois nós, com índice inteiro, já existentes, mas antes esse número incrementado de uma unidade.

Com este mecanismo de atribuição de nós, sem postos, entre quaisquer dois nós, com índice inteiro, das linhas Hidro, torna possível definir-se uma política de numeração de postos comum a todas as linhas existentes na fábrica, independentemente das especificidades das várias linhas de transporte que podem determinar, por exemplo, a introdução de conjuntos de transfers adicionais, para satisfazer as necessidades particulares de cada linha Hidro.

B.1.2 Linhas de transporte inferior

Tal como no caso das linhas de transporte superior, também para as linhas de transporte inferior se pode conceber uma planta genérica (fig. B.4) e designá-la de *nó básico inferior*, visto que a partir desta planta genérica é possível obter praticamente toda a parte inferior das linhas Hidro.

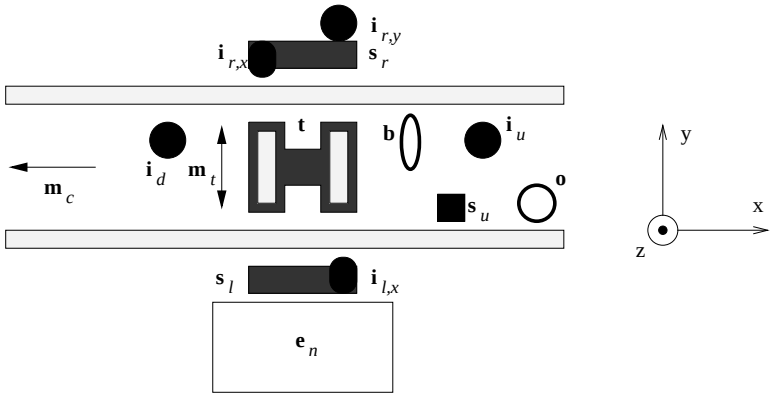


Figura B.4: Planta genérica dum nó básico inferior.

Um nó básico inferior difere dum nó básico superior, essencialmente, por conter um elevador (em vez de postos) e por o fluxo de paletes ser realizado no sentido oposto, ou seja, no sentido decrescente do eixo Ox. Além disso, existe a necessidade de incluir um sensor óptico **o** que determine a (in)existência de um auto-rádio em cima da palete detectada por $\mathbf{i}_u$.

O nó básico inferior não permite gerar directamente, a partir dos elevadores topologicamente já representados na fig. B.4, o troço de linha inferior que contém o elevador $\mathbf{e}_\lambda$, já que este elevador realiza a troca de paletes com as linhas de transporte ao longo do eixo Ox, em contraponto aos outros quatro elevadores que realizam a troca de paletes com as linhas de transporte ao longo do eixo Oy, tal como é possível constatar na fig. 7.21.

Se se replicar o nó básico inferior no eixo Oy (instanciando o parâmetro *line* com D e E, e mantendo indefinido o parâmetro n) e se se concatenarem as duas instâncias obtidas, obtém-se

um *nó composto inferior* que representa a planta genérica de uma linha HIDRO com duas linhas de transporte inferior (fig. B.5). Os ajustes a efectuar, incluem, unicamente, a eliminação, em $\mathbf{L}_E$, do elevador e dos objectos do acoplamento mecânico $\mathbf{s}_{E,l}\mathbf{i}_{E,l,x}$ e a eliminação, em $\mathbf{L}_D$, do sensor $\mathbf{i}_{D,l,x}$.

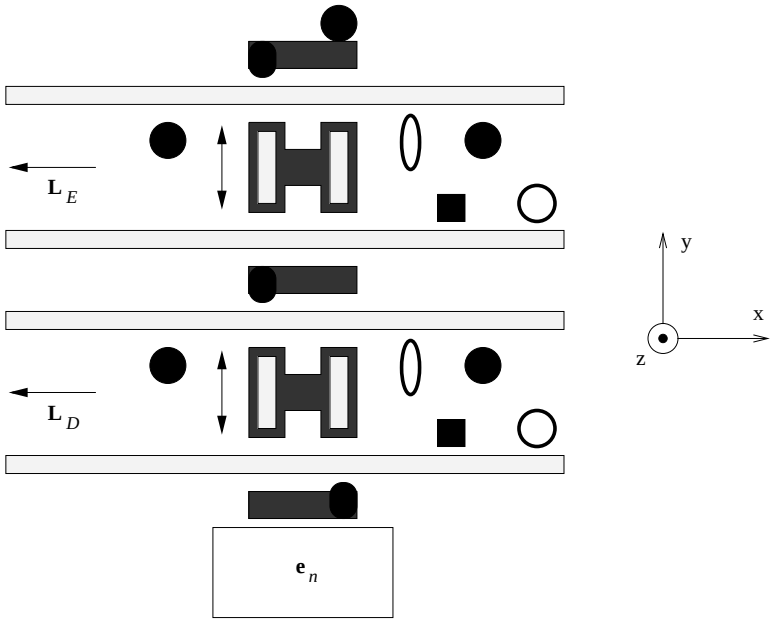


Figura B.5: Planta genérica dum nó composto inferior.

Observa-se novamente a existência de uma excepção, desta vez na incapacidade do nó composto inferior em representar o troço de linha inferior que contém o elevador $\mathbf{e}_\alpha$, visto que este elevador realiza troca de paletes com $\mathbf{L}_B$ e $\mathbf{L}_E$, em vez de $\mathbf{L}_C$ e $\mathbf{L}_D$, como acontece com $\mathbf{e}_\beta$, $\mathbf{e}_\delta$ e $\mathbf{e}_\varepsilon$. Assim, a representação do troço de linha inferior que contém o elevador $\mathbf{e}_\alpha$ é conseguida instanciando directamente o nó básico inferior em $\mathbf{L}_E$, não havendo a necessidade de recorrer a um nó composto inferior.

Os elevadores representados na fig. 7.21 correspondem às implementações actuais das linhas HIDRO, quer no que concerne à quantidade, quer no que diz respeito à sua localização (relativa às zonas de processamento). No entanto, a generalização de tais parâmetros (quantidade e localização) pode tornar-se vantajosa para possibilitar tratar situações particulares futuras em que alguma linha HIDRO venha a necessitar de adicionar novos elevadores (ou, eventualmente, conjuntos de transfers sem elevadores). Desta forma, também para as linhas de transporte inferior, tal como para as linhas de transporte superior, com a excepção dos nós com postos, é permitido utilizar índices de nó (parâmetros n) não inteiros.

Existem três situações possíveis para os nós das linhas de transporte inferior, exemplificadas na fig. B.6:

1. O nó inferior possui elevador e este está ligado a um nó superior com postos. Nesta situação, o nó inferior e o elevador têm o índice n igual ao do nó superior, que corresponde ao índice (inteiro) do postos ($n = 8$ na fig. B.6);
2. O nó inferior possui elevador, mas este não está ligado a um nó superior com postos. Neste caso, o nó inferior e o elevador têm o índice n igual ao do nó superior, em princípio fraccionário composto ($n = 7\frac{3}{4}$ na fig. B.6);

3. O nó inferior não possui elevador, pelo que, nesta situação, o respectivo índice n deve ser calculado com base nos índices dos nós que nas linhas superiores se encontram a limitar fisicamente a projecção, na linha superior, do nó inferior em causa ($n \in]7\frac{1}{4}, 7\frac{2}{4}[$ na fig. B.6).

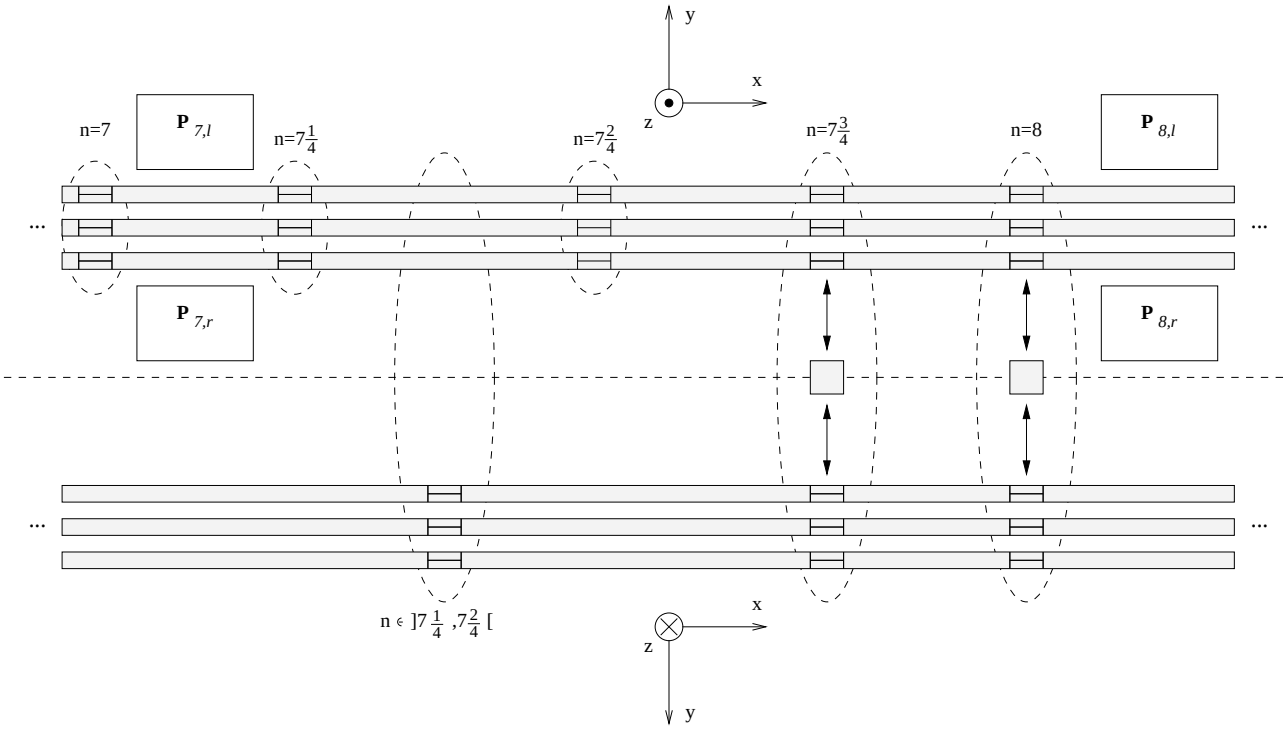


Figura B.6: Adição de nós inferiores às linhas HIDRO.

B.1.3 Elevadores

Seguindo a generalização efectuada anteriormente para a definição dos índices dos nós inferiores, a fórmula genérica para definir elevadores será $\mathbf{e}_n : n \in [0, \mathbf{k}_p + 1[$, com $\alpha = 0$ e $\lambda = \lim_{n \to \mathbf{k}_p+1} n$.

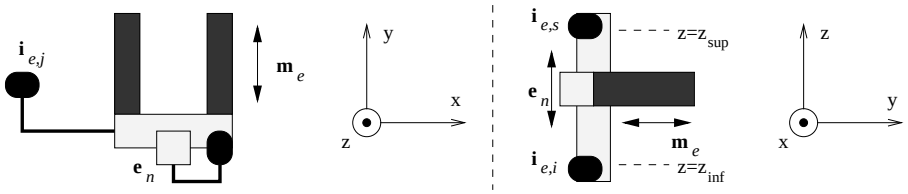
Retomando as implementações actuais das linhas HIDRO, para os elevadores $\mathbf{e}_\alpha$, $\mathbf{e}_\beta$, $\mathbf{e}_\delta$ e $\mathbf{e}_\varepsilon$, pode definir-se uma planta genérica (fig. B.7), onde se reconhecem os seguintes objectos do sistema controlado:

Elevador $\mathbf{e}_n$: Este actuador realiza deslocamentos ao longo do eixo Oz, estabelecendo as ligações (transporte de paletes) entre as linhas de transporte superior e as linhas de transporte inferior.

Passadeiras do elevador $\mathbf{m}_e$: Este actuador (acoplado à parte móvel do elevador) permite deslocar uma palete, ao longo do eixo Oy, de modo a introduzi-la no elevador (recebendo-a de uma linha de transporte adjacente), ou a retirá-la do elevador (fornecendo-a a uma linha de transporte adjacente).

Indutivo horizontal $\mathbf{i}_{e,j}$: Este sensor (acoplado à parte móvel do elevador) permite verificar a (in)existência de uma palete em cima das passadeiras $\mathbf{m}_e$ (elevador com palete, se sensor com saída a ON).

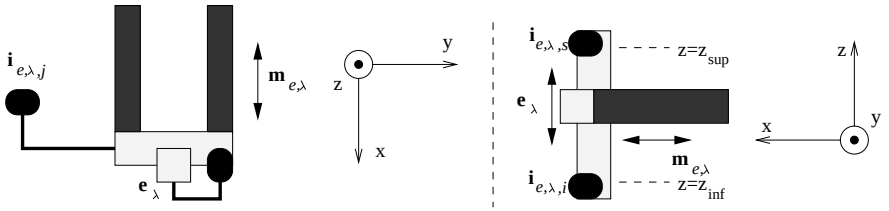
Indutivos horizontais $\mathbf{i}_{e,s}$ e $\mathbf{i}_{e,i}$: Estes sensores (acoplados à parte fixa do elevador) permitem detectar a posição, no eixo Oz, do actuador $\mathbf{e}_n$ (elevador posicionado em $z = z_{sup}$, se sensor $\mathbf{i}_{e,s}$ com saída a ON; elevador posicionado em $z = z_{inf}$, se sensor $\mathbf{i}_{e,i}$ com saída a ON).



(a) Plano xy. (b) Plano yz.

Figura B.7: Planta genérica dos elevadores $\mathbf{e}_\alpha$, $\mathbf{e}_\beta$, $\mathbf{e}_\delta$ e $\mathbf{e}_\varepsilon$.

A situação de excepção do elevador $\mathbf{e}_\lambda$ é representada pela fig. B.8. No caso deste elevador, existe, excepcionalmente, em cada uma das linhas $\mathbf{L}_B$ e $\mathbf{L}_E$, um sensor indutivo vertical $\mathbf{i}_{e,\lambda,site} \wedge site \in \{B, E\}$ e um stopper $\mathbf{s}_{e,\lambda,site} \wedge site \in \{B, E\}$, para regular a entrada e saída de paletes do elevador.



(a) Plano xy. (b) Plano xz.

Figura B.8: Planta do elevador $\mathbf{e}_\lambda$.

Na descrição das linhas de transporte superior, não se fez referência ao modo como se representam os nós superiores com elevadores. Neste momento, em que as relações entre os elevadores e as linhas de transporte inferiores já foram suficientemente bem caracterizadas, deve ser óbvio que o modo de conseguir tal representação consiste em concatenar ao extremo do eixo Oy (junto a $\mathbf{s}_r$) do nó superior (fig. B.2) a planta genérica do elevador [fig. B.7(a)], no caso de $\mathbf{e}_\alpha$, $\mathbf{e}_\beta$, $\mathbf{e}_\delta$ e $\mathbf{e}_\varepsilon$. Para $\mathbf{e}_\alpha$ e $\mathbf{e}_\lambda$ devem eliminar-se os postos do nó superior; neste último caso, a concatenação da planta do elevador $\mathbf{e}_\lambda$ deve ser realizada ao extremo de condução do eixo Ox.

B.1.4 Robô

O robô (fig. B.9) consiste no sistema controlado mais complexo das linhas HIDRO, uma vez que possui quatro graus de liberdade, correspondendo cada um deles a um actuador diferente:

Calha robótica $\mathbf{m}_{h,x}$: Este actuador executa movimentos ao longo do eixo Ox (horizontais), deslocando o braço robótico desde o local onde os sub-processos anteriores depositam os auto-rádios ($x = x_{bck}$) até um local de $\mathbf{L}_B$, imediatamente a seguir a $\mathbf{e}_\alpha$ ($x = x_{frt}$).

Braço robótico $\mathbf{m}_{h,z}$: Este actuador executa movimentos ao longo do eixo Oz (verticais), permitindo subir ($z = z_{hup}$) e descer ($z = z_{hdn}$) o braço robótico para afastar e aproximar a mão robótica dos auto-rádios.

Mão robótica $\mathbf{m}_{h,xy}$: Este actuador executa movimentos ao longo dos eixos Ox e Oy (horizontais), permitindo abrir (distância d_2 entre os dedos) e fechar (distância d_1 entre os dedos) a mão robótica para pegar e largar os auto-rádios.

Mão robótica $\mathbf{m}_{h,\theta}$: Este actuador executa movimentos ao longo da direcção do versor $O\theta$ (angulares), permitindo rodar a mão robótica para manipular correctamente os auto-rádios.

Mola $\mathbf{m}_{h,zz}$: Este actuador executa movimentos ao longo da direcção do eixo Oz, segurando a palete que se encontra em $x = x_{frt}$ para que o robô possa lá colocar um auto-rádio.

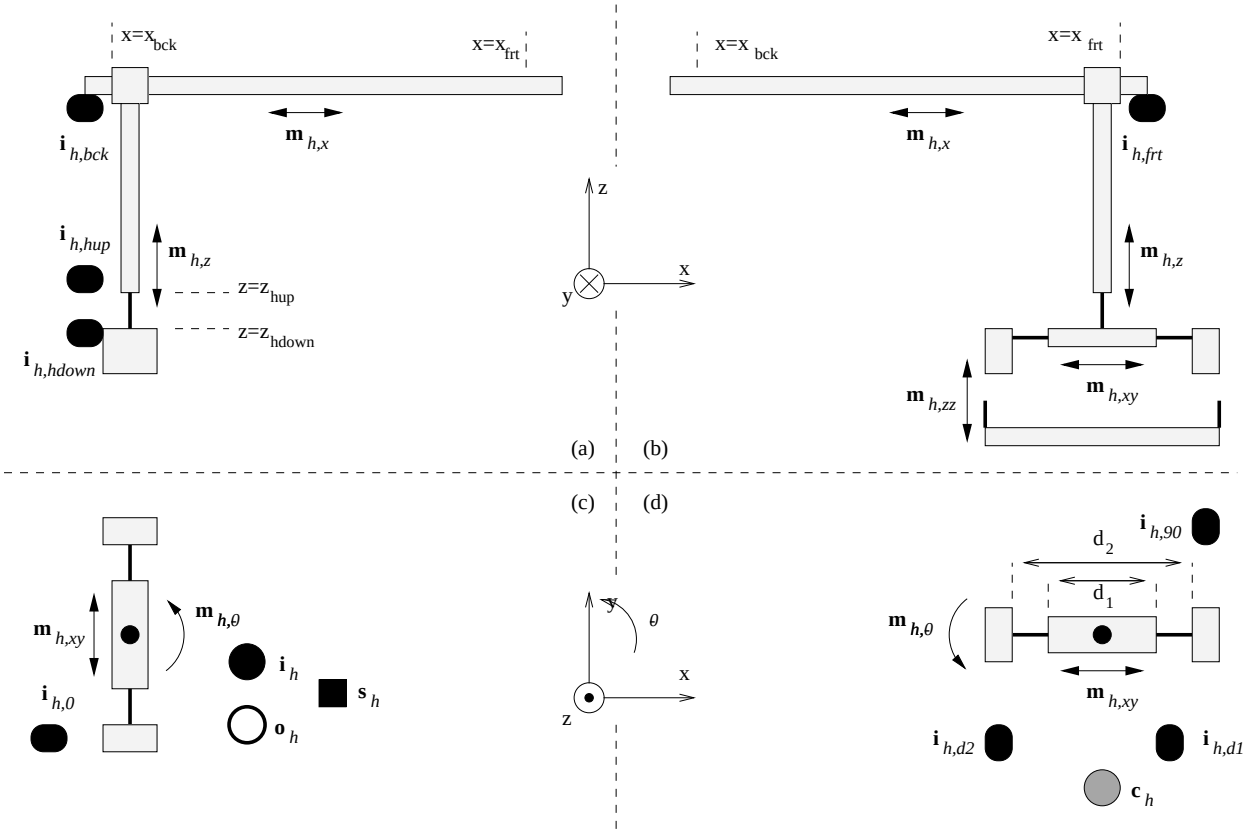


Figura B.9: Plantas do robô: (a) Calha em $x = x_{bck}$; (b) Calha em $x = x_{frt}$; (c) Mão em $\theta = 0$; (d) Mão em $\theta = 90^o$.

As fig. B.9 (c) e (d) não apresentam os actuadores $\mathbf{m}_{h,x}$, $\mathbf{m}_{h,z}$ e $\mathbf{m}_{h,zz}$, mas somente o bloco actuador $\mathbf{m}_{h,xy}\mathbf{m}_{h,\theta}$. Para além dos quatro actuadores já descritos, existem ainda os seguintes objectos:

Indutivos horizontais $\mathbf{i}_{h,bck}$ e $\mathbf{i}_{h,frt}$: Estes sensores permitem detectar a presença do braço robótico nos dois extremos da calha robótica (braço robótico posicionado em $x = x_{bck}$, se sensor $\mathbf{i}_{h,bck}$ com saída a ON; braço robótico posicionado em $x = x_{frt}$, se sensor $\mathbf{i}_{h,frt}$ com saída a ON).

Indutivos horizontais $\mathbf{i}_{h,hup}$ e $\mathbf{i}_{h,hdn}$: Estes sensores permitem detectar a posição do braço robótico nos dois extremos verticais (braço robótico posicionado em $z = z_{hup}$, se sensor $\mathbf{i}_{h,hup}$ com saída a ON; braço robótico posicionado em $z = z_{hdn}$, se sensor $\mathbf{i}_{h,hdn}$ com saída a ON).

Indutivos horizontais $\mathbf{i}_{h,0}$ e $\mathbf{i}_{h,90}$: Estes sensores permitem detectar a posição da mão robótica nos dois extremos angulares (mão robótica posicionada em $\theta = 0$, se sensor $\mathbf{i}_{h,0}$ com saída a

ON; mão robótica posicionada em $\theta = 90^o$, se sensor $\mathbf{i}_{h,90}$ com saída a ON).

Indutivos horizontais $\mathbf{i}_{h,d_1}$ e $\mathbf{i}_{h,d_2}$: Estes sensores permitem detectar a posição dos dedos da mão robótica (mão robótica fechada, se sensor $\mathbf{i}_{h,d_1}$ com saída a ON; mão robótica aberta, se sensor $\mathbf{i}_{h,d_2}$ com saída a ON).

Capacitivo $\mathbf{c}_h$: Este sensor permite detectar a existência de um auto-rádio em $x = x_{bck}$ para ser transportado pelo robô (auto-rádio em $x = x_{bck}$, se sensor $\mathbf{c}_h$ com saída a ON).

Indutivo vertical $\mathbf{i}_h$: Este sensor permite detectar a presença de uma palete em $x = x_{frt}$ para que o robô lhe possa colocar um auto-rádio em cima (palete em $x = x_{frt}$, se sensor $\mathbf{i}_h$ com saída a ON).

Óptico $\mathbf{o}_h$: Apesar de representado na fig. B.9 (c), este sensor localiza-se no interior do elevador $\mathbf{e}_\alpha$, permitindo verificar a (in)existência de um auto-rádio em cima da palete que se encontre no seu interior, para que em $x = x_{frt}$ não se coloque um auto-rádio em cima de uma palete que já possui um auto-rádio em cima (palete sem auto-rádio, se sensor $\mathbf{o}_h$ com saída a ON).

Stopper $\mathbf{s}_h$: Este actuador permite manter uma palete em $x = x_{frt}$ para que o robô lhe possa colocar um auto-rádio em cima.

Por uma questão de flexibilidade, quer no que diz respeito ao nível de abstracção em que se especificam os ciclos de controlo, quer relativamente à tecnologia de implementação dos quatro principais actuadores do robô ($\mathbf{m}_{h,x}$, $\mathbf{m}_{h,z}$, $\mathbf{m}_{h,xy}$ e $\mathbf{m}_{h,\theta}$), é possível, de uma forma consciente, recorrer a um de dois tipos de controlo do robô: (1) malha aberta, situação em que os quatro actuadores são utilizados tal como descritos na tabela 3.2, dispensando os oito sensores indutivos horizontais ($\mathbf{i}_{h,bck}$, $\mathbf{i}_{h,frt}$, $\mathbf{i}_{h,hup}$, $\mathbf{i}_{h,hdn}$, $\mathbf{i}_{h,0}$, $\mathbf{i}_{h,90}$, $\mathbf{i}_{h,d_1}$, $\mathbf{i}_{h,d_2}$); (2) malha fechada, situação em que é necessário utilizar a informação fornecida pelos oito sensores indutivos horizontais e em que os comandos disponibilizados pelos quatro actuadores do robô limitam-se a executar cegamente os movimentos até que um comando de paragem seja enviado.

B.2 Tabelas de sensores e actuadores

Nesta secção, apresentam-se várias tabelas que descrevem os vários elementos que compõem o sistema controlado do SCLH.

B.2.1 Sensores

Símbolo	Notação	Caracterização	Estado
 indutivo vertical	$\mathbf{i}_{line,n,site}$ - $line \in \{A, B, C\}$ - $n \in [0, \mathbf{K}_p + 1[$ - $site \in \{u, d, p\}$	1. sensor indutivo; 2. sem trigger; 3. saída de dois estados (On e Off); 4. saída normalmente a Off; 5. situado nas linhas superiores.	1. a saída passa a On, quando detecta uma palete; 2. a saída está a Off, em todas as outras situações.
	$\mathbf{i}_{line,n,site}$ - $line \in \{D, E, F\}$ - $n \in [0, \mathbf{K}_p + 1[$ - $site \in \{u, d\}$	1 a 4. igual a $\mathbf{i}_{line,n,site}$: $line \in \{A, B, C\} \wedge site \in \{u, d, p\}$; 5. situado nas linhas inferiores.	1 e 2. igual a $\mathbf{i}_{line,n,site}$: $line \in \{A, B, C\} \wedge site \in \{u, d, p\}$.
	$\mathbf{i}_{line,n,site,dir}$ - $line \in \{A \ldots F\}$ - $n \in [0, \mathbf{K}_p + 1[$ - $site \in \{l, r\}$ - $dir = y$	1 a 4. igual a $\mathbf{i}_{line,n,site}$: $line \in \{A, B, C\} \wedge site \in \{u, d, p\}$;	1 e 2. igual a $\mathbf{i}_{line,n,site}$: $line \in \{A, B, C\} \wedge site \in \{u, d, p\}$.
	$\mathbf{i}_{line,n,site,dir}$ - $line = \mathbf{e}$ - $n = \lambda$ - $site \in \{B, E\}$	1 a 4. igual a $\mathbf{i}_{line,n,site}$: $line \in \{A, B, C\} \wedge site \in \{u, d, p\}$;	1 e 2. igual a $\mathbf{i}_{line,n,site}$: $line \in \{A, B, C\} \wedge site \in \{u, d, p\}$.
 leitor	$\mathbf{b}_{line,n}$ - $line \in \{A \ldots F\}$ - $n \in [0, \mathbf{K}_p + 1[$	1. leitor de código de barras; 2. com trigger; 3. saída de dois estados (Id e X); 4. saída normalmente a X; 5. localizado nas linhas.	1. Quando detecta a presença dum auto-rádio, coloca na saída o respectivo Id; 2. a saída está a X, em todas as outras situações.
	$\mathbf{b}_{line,n}$ - $line = \mathbf{p}$ - $n \in \{1 \ldots \mathbf{K}_p\}$	1 a 4. igual a $\mathbf{b}_{line,n}$: $line \in \{A \ldots F\}$; 5. localizado nos postos.	1 e 2. igual a $\mathbf{b}_{line,n}$: $line \in \{A \ldots F\}$.

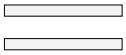
Símbolo	Notação	Caracterização	Estado
 indutivo horizontal	$\mathbf{i}_{line,n,site,dir}$ - $line \in \{A \ldots F\}$ - $n \in [0, \mathbf{K}_p + 1[$ - $site \in \{l, r\}$ - $dir = x$	1 a 4. igual a $\mathbf{i}_{line,n,site}$: $line \in \{A, B, C\} \wedge site \in \{u, d, p\}$; 5. situado nas linhas superiores; 6. pode estar mecanicamente acoplado a um actuador do tipo $\mathbf{s}_{line,n,site}$.	1 e 2. igual a $\mathbf{i}_{line,n,site}$: $line \in \{A, B, C\} \wedge site \in \{u, d, p\}$.
	$\mathbf{i}_{line,n,site}$ - $line = e$ - $n \in [0, \mathbf{K}_p + 1[$ - $site = j$	1 a 4. igual a $\mathbf{i}_{line,n,site}$: $line \in \{A, B, C\} \wedge site \in \{u, d, p\}$; 5. pode estar mecanicamente acoplado à parte móvel dum actuador do tipo $\mathbf{e}_n$.	1. a saída passa a ON, quando detecta uma palete no interior do elevador; 2. a saída está a OFF, em todas as outras situações.
	$\mathbf{i}_{line,n,site}$ - $line = e$ - $n \in [0, \mathbf{K}_p + 1[$ - $site \in \{s, i\}$	1 a 4. igual a $\mathbf{i}_{line,n,site}$: $line \in \{A, B, C\} \wedge site \in \{u, d, p\}$; 5. mecanicamente acoplado à parte fixa dum actuador do tipo $\mathbf{e}_n$.	1. a saída passa a ON, quando detecta a parte móvel do elevador; 2. a saída está a OFF, em todas as outras situações.
	$\mathbf{i}_{line,site}$ - $line = h$ - $site \in \{bck, frt\}$	1 a 4. igual a $\mathbf{i}_{line,n,site}$: $line \in \{A, B, C\} \wedge site \in \{u, d, p\}$; 5. mecanicamente acoplado à calha robótica.	- a saída passa a ON, quando detecta o braço robótico $\mathbf{m}_{r,z}$ em $x = x_{bck}$ (para $site = bck$), ou em $x = x_{frt}$ (para $site = frt$); - a saída está a OFF, em todas as outras situações.
	$\mathbf{i}_{line,site}$ - $line = h$ - $site \in \{hup, hdn\}$	1 a 4. igual a $\mathbf{i}_{line,n,site}$: $line \in \{A, B, C\} \wedge site \in \{u, d, p\}$; 5. mecanicamente acoplado ao braço robótico.	- a saída passa a ON, quando detecta a mão robótica $\mathbf{m}_{r,xy}$ em $z = z_{hup}$ (para $site = hup$), ou em $z = z_{hdn}$ (para $site = hdn$); - a saída está a OFF, em todas as outras situações.
	$\mathbf{i}_{line,site}$ - $line = h$ - $site \in \{0, 90\}$	1 a 4. igual a $\mathbf{i}_{line,n,site}$: $line \in \{A, B, C\} \wedge site \in \{u, d, p\}$; 5. mecanicamente acoplado ao braço robótico.	- a saída passa a ON, quando detecta a mão robótica $\mathbf{m}_{r,xy}$ em $\theta = 0$ (para $site = 0$), ou em $\theta = 90^o$ (para $site = 90$); - a saída está a OFF, em todas as outras situações.
	$\mathbf{i}_{line,site}$ - $line = h$ - $site \in \{d1, d2\}$	1 a 4. igual a $\mathbf{i}_{line,n,site}$: $line \in \{A, B, C\} \wedge site \in \{u, d, p\}$; 5. mecanicamente acoplado à mão robótica.	- a saída passa a ON, quando detecta os dedos da mão robótica com uma distância entre eles de d_1 ou d_2 (para $site = d1$ ou $site = d2$); - a saída está a OFF, em todas as outras situações.

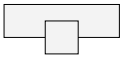
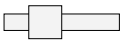
Símbolo	Notação	Caracterização	Estado
 óptico	$\mathbf{o}_{line,n}$ - $line \in \{D, E, F\}$ - $n \in [0, \mathbf{K}_p + 1[$	1. sensor óptivo; 2. sem trigger; 3. saída de dois estados (On e Off); 4. saída normalmente a On; 5. situado nas linhas inferiores.	1. a saída passa a Off, quando detecta um objecto (normalmente um auto-rádio); 2. a saída está a On, em todas as outras situações.
	$\mathbf{o}_{site}$ - $site = h$	1 a 4. igual a $\mathbf{o}_{line,n}$: $line \in \{D, E, F\}$; 5. localizado no interior de $\mathbf{e}_\alpha$.	1 e 2. igual a $\mathbf{o}_{line,n}$: $line \in \{D, E, F\}$.
 capacitivo	$\mathbf{c}_{site}$ - $site = h$	1. sensor capacitivo; 2. sem trigger; 3. saída de dois estados (On e Off); 4. saída normalmente a Off; 5. localizado em $x = x_{frt}$.	1. Quando detecta a presença dum auto-rádio em $x = x_{bck}$, a saída passa a Off; 2. a saída está a On, em todas as outras situações.

B.2.2 Actuadores

Símbolo	Notação	Caracterização	Estado
 stopper	$\mathbf{s}_{line,n,site}$ - $line \in \{A,B,C\}$ - $n \in [0, \mathbf{K}_p + 1[$ - $site \in \{u,p\}$	1. actuador pneumático; 2. recebe dois comandos (UP e DOWN); 3. posiciona-se no eixo Oz em duas posições ($z = z_{up}$ e $z = z_{down}$); 4. posição normal em $z = z_{up}$; 5. situado nas linhas superiores.	1. posiciona-se em $z = z_{up}$, quando recebe o comando UP, impedindo a passagem de paletes; 2. posiciona-se em $z = z_{down}$, quando recebe o comando DOWN, permitindo a passagem de paletes.
	$\mathbf{s}_{line,n,site}$ - $line \in \{D,E,F\}$ - $n \in [0, \mathbf{K}_p + 1[$ - $site = u$	1 a 4. igual a $\mathbf{s}_{line,n,site}$: $line \in \{A,B,C\} \wedge site \in \{u,p\}$; 5. situado nas linhas inferiores.	1 e 2. igual a $\mathbf{i}_{line,n,site}$: $line \in \{A,B,C\} \wedge site \in \{u,p\}$.
	$\mathbf{s}_{line,n,site}$ - $line = \mathbf{e}$ - $n = \lambda$ - $site \in \{B,E\}$	1 a 4. igual a $\mathbf{i}_{line,n,site}$: $line \in \{A,B,C\} \wedge site \in \{u,p\}$.	1 e 2. igual a $\mathbf{i}_{line,n,site}$: $line \in \{A,B,C\} \wedge site \in \{u,p\}$.
	$\mathbf{s}_{site}$ - $site = h$	1 a 4. igual a $\mathbf{i}_{line,n,site}$: $line \in \{A,B,C\} \wedge site \in \{u,p\}$; 5. localizado em $x = x_{frt}$.	1 e 2. igual a $\mathbf{i}_{line,n,site}$: $line \in \{A,B,C\} \wedge site \in \{u,p\}$.
 batente	$\mathbf{s}_{line,n,site}$ - $line \in \{A \ldots F\}$ - $n \in [0, \mathbf{K}_p + 1[$ - $site \in \{l,r\}$	1 a 4. igual a $\mathbf{s}_{line,n,site}$: $line \in \{A,B,C\} \wedge site \in \{u,p\}$. 5. sob controlo, não possui mais nenhum tipo de movimentos; 6. pode estar mecanicamente acoplado a um sensor do tipo $\mathbf{i}_{line,n,site,dir}$: $line \in \{A \ldots F\} \wedge site \in \{l,r\} \wedge dir = x$.	1. posiciona-se em $z = z_{up}$, quando recebe o comando UP (impedindo a passagem de paletes) e desde que se trate dum actuador que não está situado num extremo de condução do eixo Oy; caso contrário, não responde a comandos, i.e. não é controlado; 2. posiciona-se em $z = z_{down}$, quando recebe o comando DOWN (permitindo a passagem de paletes) e desde que se trate de um actuador que não está situado num extremo de condução do eixo Oy; caso contrário, não responde a comandos, i.e. não é controlado.

Símbolo	Notação	Caracterização	Estado
 transfer	$\mathbf{t}_{line,n}$ - $line \in \{A \ldots F\}$ - $n \in [0, \mathbf{K}_p + 1[$	1. actuador pneumático; 2. recebe três comandos (Up, Middle e Down); 3. posiciona-se no eixo Oz em três posições ($z = z_{up}$, $z = z_{middle}$ e $z = z_{down}$); 4. posição normal em $z = z_{middle}$; 5. mecanicamente acoplado a um sensor do tipo $\mathbf{m}_{t,line,n}$.	1. posiciona-se em $z = z_{up}$, quando recebe o comando Up, possibilitando a transferência de paletes por actuadores do tipo $\mathbf{m}_{t,line,n}$; 2. posiciona-se em $z = z_{down}$, quando recebe o comando Down, permitindo o transporte de paletes por actuadores do tipo $\mathbf{m}_{c,line}$; 3. situa-se em $z = z_{middle}$, quando recebe o comando Middle, impedindo a passagem de paletes.
 passadeiras de transfers	$\mathbf{m}_{type,line,n}$ - $type = t$ - $line \in \{A, B, C\}$ - $n \in [0, \mathbf{K}_p + 1[$	1. actuador electro-mecânico; 2. recebe três comandos (Left, Right e Stop); 3. movimenta as passadeiras nos dois sentidos do eixo Oy; 4. estado das passadeiras normalmente parado; 5. sob controlo directo, não possui mais nenhum tipo de movimentos; 6. situado nas linhas superiores; 7. mecanicamente acoplado a um actuador do tipo $\mathbf{t}_{line,n} : line \in \{A, B, C\}$.	1. desloca as passadeiras no sentido crescente do eixo Oy, quando recebe o comando Left, transportando a palete para a esquerda; 2. desloca as passadeiras no sentido decrescente do eixo Oy, quando recebe o comando Right, transportando a palete para a direita; 3. pára o movimento das passadeiras, quando recebe o comando Stop.
	$\mathbf{m}_{type,line,n}$ - $type = t$ - $line \in \{D, E, F\}$ - $n \in [0, \mathbf{K}_p + 1[$	1 a 5. igual a $\mathbf{m}_{t,line,n,site} \wedge line \in \{A, B, C\}$; 6. situado nas linhas inferiores; 7. mecanicamente acoplado a um actuador do tipo $\mathbf{t}_{line,n} : line \in \{D, E, F\}$.	1. desloca as passadeiras no sentido decrescente do eixo Oy, quando recebe o comando Left, transportando a palete para a esquerda; 2. desloca as passadeiras no sentido crescente do eixo Oy, quando recebe o comando Right, transportando a palete para a direita; 3. pára o movimento das passadeiras, quando recebe o comando Stop.

Símbolo	Notação	Caracterização	Estado
 passadeiras de condução	$\mathbf{m}_{type,line}$ - $type = c$ - $line \in \{A \ldots F\}$	1. actuador electro-mecânico; 2. recebe três comandos (FWD, BWD e STOP); 3. movimenta as passadeiras nos dois sentidos do eixo Ox; 4. normalmente em funcionamento.	1. desloca as passadeiras no sentido crescente do eixo Ox, quando recebe o comando FWD (as passadeiras das linhas superiores $\mathbf{L}_A$, $\mathbf{L}_B$ e $\mathbf{L}_C$ apresentam um movimento permanente neste sentido, ou seja, comando BWD nunca utilizado); 2. desloca as passadeiras no sentido decrescente do eixo Ox, quando recebe o comando BWD (as passadeiras das linhas inferiores $\mathbf{L}_D$, $\mathbf{L}_E$ e $\mathbf{L}_F$ apresentam um movimento permanente neste sentido, ou seja, comando FWD nunca utilizado); 3. pára o movimento das passadeiras, quando recebe o comando STOP.
 passadeiras de elevadores	$\mathbf{m}_{type,n}$ - $type = e$ - $n \in \{\alpha, \beta, \delta, \varepsilon\}$	1. actuador electro-mecânico; 2. recebe três comandos (IN, OUT e STOP); 3. movimenta as passadeiras nos dois sentidos do eixo Oy; 4. estado das passadeiras normalmente parado; 5. sob controlo directo, não possui mais nenhum tipo de movimentos; 6. mecanicamente acoplado a um actuador do tipo $\mathbf{e}_n$.	1. desloca as passadeiras no sentido crescente do eixo Oy, quando recebe o comando IN, transportando a palete para o interior do elevador; 2. desloca as passadeiras no sentido decrescente do eixo Oy, quando recebe o comando OUT, transportando a palete para o exterior do elevador; 3. pára o movimento das passadeiras, quando recebe o comando STOP.
	$\mathbf{m}_{type,n}$ - $type = e$ - $n = \lambda$	1, 2, 4 a 6. igual a $\mathbf{m}_{e,n}$: $nn \in \{\alpha, \beta, \delta, \varepsilon\}$; 3. movimenta as passadeiras nos dois sentidos do eixo Ox;	1. desloca as passadeiras no sentido crescente do eixo Ox, quando recebe o comando IN, transportando a palete para o interior do elevador; 2. desloca as passadeiras no sentido decrescente do eixo Oxy, quando recebe o comando OUT, transportando a palete para o exterior do elevador; 3. pára o movimento das passadeiras, quando recebe o comando STOP.

Símbolo	Notação	Caracterização	Estado
 elevador	$\mathbf{e}_n$ - $e \in [0, \mathbf{K_p} + 1]$	1. actuador electro-mecânico; 2. recebe dois comandos (Inf e Sup); 3. posiciona-se no eixo Oz em duas posições ($z = z_{inf}$ e $z = z_{sup}$); 4. mecanicamente acoplado a um actuador do tipo $\mathbf{m}_{e,n}$ e a três sensores do tipo $\mathbf{m}_{e,n,site}$.	1. posiciona-se em $z = z_{sup}$, quando recebe o comando Sup, possibilitando a transferência de paletes de e para as linhas de transporte superior por actuadores do tipo $\mathbf{m}_{e,n}$; 2. posiciona-se em $z = z_{inf}$, quando recebe o comando Inf, possibilitando a transferência de paletes de e para as linhas de transporte inferior por actuadores do tipo $\mathbf{m}_{e,n}$.
 calha robótica	$\mathbf{m}_{type,dir}$ - $type = h$ - $dir = x$	1. actuador electro-mecânico; 2. recebe dois comandos (Fwd e Bwd); 3. posiciona-se no eixo Ox em duas posições ($x = x_{bck}$ e $x = x_{frt}$); 4. posição normal em $x = x_{frt}$; 5. mecanicamente acoplado a um actuador do tipo $\mathbf{m}_{r,z}$.	1. posiciona-se em $x = x_{bck}$, quando recebe o comando Bwd, permitindo que o braço robótico $\mathbf{m}_{r,z}$ possa recolher o auto-rádio; 2. posiciona-se em $x = x_{frt}$, quando recebe o comando Fwd, permitindo que o braço robótico $\mathbf{m}_{r,z}$ possa depositar o auto-rádio.
 braço robótico	$\mathbf{m}_{type,dir}$ - $type = h$ - $dir = z$	1. actuador pneumático; 2. recebe dois comandos (Up e Down); 3. posiciona-se no eixo Oz em duas posições ($z = z_{hup}$ e $x = x_{hdn}$); 4. posição normal em $z = z_{hup}$; 5. sob controlo directo, não possui mais nenhum movimento; 6. mecanicamente acoplado a um actuador do tipo $\mathbf{m}_{r,z}$ e ao bloco actuador $\mathbf{m}_{r,xy}\mathbf{m}_{r,\theta}$.	1. posiciona-se em $z = z_{hup}$, quando recebe o comando Up, subindo o braço robótico $\mathbf{m}_{r,z}$; 2. posiciona-se em $z = z_{hdn}$, quando recebe o comando Down, subindo o braço robótico $\mathbf{m}_{r,z}$.

Símbolo	Notação	Caracterização	Estado
 mão robótica	$\mathbf{m}_{type,dir}$ - $type = h$ - $dir = xy$	1. actuador pneumático; 2. recebe dois comandos (OPEN e CLOSE); 3. posiciona os dois dedos da mão robótica a distâncias entre eles de (d_1 e d_2); 4. distância normal de d_2; 5. sob controlo directo, não possui mais nenhum movimento; 6. mecanicamente acoplado a um actuador do tipo $\mathbf{m}_{r,z}$ e parte integrante do bloco $\mathbf{m}_{r,xy}\mathbf{m}_{r,\theta}$.	1. posiciona os dois dedos da mão robótica a uma distância d_2 entre eles, quando recebe o comando OPEN, abrindo a mão robótica; 2. posiciona os dois dedos da mão robótica a uma distância d_1 entre eles, quando recebe o comando CLOSE, fechando a mão robótica.
	$\mathbf{m}_{type,dir}$ - $type = h$ - $dir = \theta$	1. actuador electro-mecânico; 2. recebe dois comandos (GET e PUT); 3. posiciona a mão robótica, seguindo a direcção angular do versor $O\theta$, em duas posições ($\theta = 0$ e $\theta = 90^o$); 4. posição normal em $\theta = 0$; 5. sob controlo directo, não possui mais nenhum movimento; 6. mecanicamente acoplado a um actuador do tipo $\mathbf{m}_{r,z}$ e parte integrante do bloco $\mathbf{m}_{r,xy}\mathbf{m}_{r,\theta}$.	1. roda a mão robótica para $\theta = 0$, quando recebe o comando GET, ajustando a mão robótica para pegar no auto-rádio; 2. roda a mão robótica para $\theta = 90^o$, quando recebe o comando PUT, ajustando a mão robótica para largar o auto-rádio.
 mola	$\mathbf{m}_{type,dir}$ - $type = h$ - $dir = zz$	1. actuador pneumático; 2. recebe dois comandos (UP e DOWN); 3. posiciona-se no eixo Oz em duas posições ($z = z_{up}$ e $z = z_{dn}$); 4. posição normal em $z = z_{dn}$.	1. posiciona-se em $z = z_{up}$, quando recebe o comando UP, prendendo a palete; 2. posiciona-se em $z = z_{dn}$, quando recebe o comando DOWN, libertando a palete.

O parâmetro *site*, qualificador dos vários objectos do sistema controlado (sensores, actuadores e postos), pode tomar os seguintes valores, tendo como referência o sentido de fluxo de auto-rádios das linhas em que se situa o objecto qualificado:

- l: quando o objecto se situa na margem esquerda (*left*) da linha;
- r: quando o objecto se situa na margem direita (*right*) da linha;
- u: quando o objecto se situa a montante (*up*) do transfer do nó em causa;
- d: quando o objecto se situa a jusante (*down*) do transfer do nó em causa.

Os seguintes símbolos são utilizados em diversos parâmetros:

- s: superior;
- i: inferior;
- e: elevador;
- h: mão (hand) robótica.
- t: transfer.

Apêndice C

Glossário

Abordagem operacional: Forma de modelação em que se utilizam especificações executáveis.

Abstracção: Mecanismo de modelação que consiste na selecção dos aspectos essenciais duma entidade, ignorando aqueles tidos por irrelevantes.

Acção: Processo instantâneo, no sentido de ocorrer muito rapidamente e de não ser susceptível de interrupção.

Actividade: Porção de trabalho que se deve efectuar em determinado tempo. Em UML, significa uma operação mais demorada, cuja execução pode ser interrompida.

Actor: Entidade externa a um sistema mas que com ele interage.

Afectação: Tarefa em que se procede à escolha dos componentes que vão permitir a implementação do sistema.

Agregação: Forma especial de associação que especifica uma relação entre o agregado e os componentes que, física ou logicamente, contém.

Agregado: Colecção de partes que juntas formam algo novo que é mais do que a simples soma dessas partes.

Análise: Fase do processo de desenvolvimento em que se produz um modelo abstracto para descrever os aspectos fundamentais do domínio de aplicação.

Associação: Relação estrutural que descreve as ligações entre objectos.

Atributo: Informação usada para expressar uma determinada característica duma classe.

Caso de uso: Descrição dum conjunto de sequências de acções que um sistema desempenha e que produz um resultado relevante para alguns actores desse sistema.

Cenário: Sequência específica de acções que ilustram o comportamento dum sistema.

Ciclo de vida: Conjunto de actividades que vão desde o momento em que o sistema é mentalmente conceptualizado até ao instante em que ele é retirado definitivamente de uso. Representa todo o conjunto de actos válidos, realizados, durante a vida útil do sistema, com o objectivo de o idealizar, desenvolver e usar.

Classe: Mecanismo de modelação que permite definir um conjunto, possivelmente infinito, de objectos, que partilham os mesmos atributos e as mesmas operações.

Classe concreta: Classe que pode ter directamente instâncias.

Classe abstracta: Classe que não tem instâncias, sendo criada para servir de base à definição de outras classes.

Classificação: Forma de agrupar objectos numa abstracção comum, pondo em evidência as características comuns desse objectos.

Composição: Forma mais restrita de agregação, em que os componentes dum agregado são criados e destruídos por este, mas que não podem ser partilhados por outros agregados.

Concepção: Fase do processo de desenvolvimento em que, com base no modelo obtido na fase de análise, é criado um modelo que especifica os componentes que realizam uma determinada solução para o sistema.

Co-projecto: Desenvolvimento integrado de sistemas digitais, implementados com componentes de hardware e de software; abordagem unificada e cooperativa para o desenvolvimento de sistemas hardware/software, em que as opções de hardware e software podem ser consideradas em conjunto

Decomposição funcional: Processo de desenvolver uma solução para uma função, expressa como um algoritmo que conjuga uma colecção de funções mais simples.

Desenvolvimento: Conjunto das fases do ciclo de vida responsáveis pela construção do sistema, incluindo, pelo menos, a análise, a concepção e a implementação.

Encapsulamento: Técnica de modelação que consiste em separar o comportamento externo dum objecto, a que têm acesso outros objectos, dos pormenores de implementação desse objecto, que são ocultados aos outros objectos.

Equipa de projecto: Ver projectista.

Especificar: Acto que leva à criação de especificações para os modelos dum sistema (utilizando linguagens).

Especificação: Representação concreta (real) do modelo dum sistema numa dada linguagem.

Estado: Período de tempo durante o qual um sistema exibe um tipo específico de comportamento; conjunto de variáveis que retratam o modo em que um sistema se emcontra.

Estereótipo: Extensão ao meta-modelo UML que permite criar novos mecanismos de modelação, baseados em mecanismos já existentes, mas específicos para um dado problema ou contexto.

Estudo de viabilidade: Fase pré-desenvolvimento, cujo propósito consiste na avaliação dos custos e benefícios do sistema proposto.

Fase: Abstracção, ao longo do tempo, dum conjunto de actividades; conceito útil para agregar actividades e relacioná-las temporalmente.

Firmware: Software embebido em dispositivos electrónicos durante a sua manufactura.

Formalismo: Meta-modelo formal.

Gramática duma linguagem: Sintaxe, lêxico e semântica que permitem concretizar, à luz do meta-modelo da linguagem, o modelo do sistema numa representação gráfica, textual ou outra.

Hardware: Conjunto de objectos tangíveis (circuitos integrados, placas de circuito impresso, cabos, fontes de alimentação, memórias, impressoras, terminais) que constituem a parte física dum sistema computacional.

Herança: Mecanismo que facilita a construção de novas classes (subclasses) a partir de outras classes (superclasses) e permite a especialização e a generalização de componentes.

Hierarquia: Mecanismo pelo qual elementos mais específicos incorporam a estrutura e o comportamento de elementos mais genéricos.

Identidade: Natureza intrínseca dum objecto que o distingue de todos os outros.

Implementação: Fase do processo de desenvolvimento em que se realiza uma solução baseada no modelo de concepção.

Linguagem: Conjunto de todas as frases válidas que é possível construir com base na respectiva gramática.

Manutenção: Fase que corre em paralelo com a utilização do sistema e em que se procuram corrigir todas as anomalias, que não foram detectadas durante o desenvolvimento, e se pretende fazer evoluir o sistema, de modo a que continue a ser útil aos seus utilizadores.

Mensagem: Forma de comunicação entre objectos que pressupõe a execução duma dada operação.

Meta-modelo (modelo dum modelo)**:** Conjunto de elementos de composição, funcionais ou estruturais, e de regras de composição que permitem construir um modelo para um dado sistema.

Meta-modelo da linguagem: Modelo conceptual, seguido na definição da linguagem, e que implica que qualquer especificação escrita nessa linguagem segue obrigatoriamente o meta-modelo por ela imposto.

Método: Conjunto de actividades (recomendações genéricas) que organizam a execução duma dada fase do ciclo de vida; modo de prosseguir com uma dada fase de desenvolvimento. Implementação duma dada operação duma classe.

Metodologia: Abordagem metódica para o desenvolvimento dum sistema, através da selecção dum modelo do processo e dum conjunto de métodos (ou técnicas) a serem aplicados.

Modelar: Acto que permite a obtenção de modelos dum sistema.

Modelo: Representação conceptual (virtual, abstracta) dum sistema, à luz dum determinado meta-modelo.

Modelo de processo: Esquema que organiza (ordena) e relaciona a forma como as várias fases e tarefas devem ser prosseguidas ao longo do ciclo de vida do sistema. A função principal dum modelo do processo é determinar a ordem das fases envolvidas durante a existência dos sistemas e estabelecer os critérios de transição para progredir entre fases.

Modelo de processo de desenvolvimento: Esquema que organiza, ordena e relaciona a forma como as várias fases e tarefas devem ser prosseguidas ao longo do desenvolvimento do sistema.

Notação: Ver linguagem.

Objecto: Entidade individual, real ou abstracta, com uma função bem definida no domínio do problema e que pode ser reconhecida pelos dados que incorpora, pelo comportamento que exibe e pelo processamento que desempenha.

Orientação ao objecto: Utilização consciente e deliberada de objectos como critério principal de organização das abstracções de dados e dos procedimentos dos sistemas.

Padrão: Solução comum para um problema típico num dado contexto.

Paradigma: Conjunto de teorias, técnicas e métodos que, conjuntamente, representam uma forma de organizar a informação; forma de representar a realidade.

Partição hardware/software: Decisão, manual ou automática, em colocar certos componentes do sistema em hardware e outros em software.

Plataforma alvo: Conjunto de elementos físicos (hardware) interligados, que constituem uma arquitectura computacional com um dado modelo de computação e de comunicação.

Polimorfismo: Mecanismo que permite que a mesma operação possa apresentar comportamentos distintos para classes diferentes.

Princípio de substituição de Liskov: Princípio que indica que uma subclasse deve responder a todas as mensagens que as suas superclasses também respondem, ou seja, se a classe B herda de A, então uma instância de B pode sempre ser usada em todos os contextos em que instâncias de A são utilizadas. Daqui resulta a impossibilidade de uma subclasse eliminar propriedades das respectivas superclasses, podendo contudo acrescentar ou especializá-las.

Processo: Sequência de factos que conduzem a certo resultado, mais particularmente, sequência de actos ordenados para a consecução de certa finalidade. No âmbito da construção metódica de sistemas, significa o conjunto de tarefas executadas ao longo do ciclo de vida do sistema.

Processo de desenvolvimento: Sequência de actos que conduzem ao desenvolvimento dum sistema.

Projectista: Indivíduo responsável pelo projecto (ou por uma parte dele) dum dado sistema.

Projecto: Actividades anteriores à utilização prática do respectivo sistema.

Propriedade: Atributo ou operação duma dada classe ou dum dado objecto.

Simulação: Exercitação numérica das entradas do modelo dum sistema para determinar como são afectadas as suas saídas; imitação do comportamento dum sistema real, através da sua descrição e análise, dando um importante contributo na sua melhoria (se o sistema ja existir) ou auxiliando o seu projecto (caso seja um sistema a implementar).

Síntese: Processo que transforma uma descrição comportamental numa descrição física.

Sistema: Colecção de componentes inter-relacionados que actuam como um todo, para atingir um determinado objectivo. É o observador do sistema que define a fronteira deste como o seu ambiente, o que torna a definição de sistema não intrínseca a este, mas dependente do seu observador, ou seja, dos objectivos particulares em cada situação.

Sistema embebido: Sistema, contendo componentes de hardware e de software, concebido para uma aplicação específica e que se encontra incorporado num sistema mais complexo.

Software: Algoritmos (instruções descrevendo pormenorizadamente como fazer algo) e respectivas representações, nomeadamente os programas.

Subclasse: Especialização duma classe numa relação hierárquica.

Superclasse: Generalização duma classe numa relação hierárquica.

Vista do sistema: Perspectiva (total ou parcial) do sistema que se pretende representar.

Técnica: Forma de proceder para a realização duma dada actividade pertencente a um método.

Teste: Fase que pode decorrer em paralelo com o desenvolvimento dum sistema, a fim de se tentar encontrar todas as falhas daquele.

UML (Unified Modeling Language)**:** Linguagem para expressar a funcionalidade, a estrutura e as relações de sistemas complexos.

Utilização: Operação do sistema em ambiente real.

Bibliografia

[Abbott, 1983] Abbott, R. J. (1983). Program Design by Informal Descriptions. *Communications of the ACM*, 26(11):882–94.

[Agsteiner et al., 1995] Agsteiner, K., Monjau, D., e Schulze, S. (1995). Object-Oriented High-Level Modelling of System Components for the Generation of VHDL. In *Proceedings of Euro-DAC'95 with Euro-VHDL'95*, pp. 436–41, Brighton, Reino Unido. IEEE Computer Society Press.

[ANSI, 1976] ANSI (1976). *American National Standards Programming Language PL/I, ANSI X3.53-1976*. American National Standards Institute, New York, E.U.A.

[Anuff, 1996] Anuff, E. (1996). *The Java Sourcebook*. John Wiley & Sons. ISBN 0-471-14859-8.

[Armstrong e Barroca, 1996] Armstrong, J. e Barroca, L. (1996). Specification and Verification of Reactive System Behaviour: The Railroad Crossing Example. *Real Time Systems*, 10(2):143–78.

[Arnold e Gosling, 1996] Arnold, K. e Gosling, J. (1996). *The Java Programming Language*. Addison-Wesley. ISBN 0-201-63455-4.

[Baber, 1997] Baber, R. L. (1997). Comparison of Electrical "Engineering" of Heaviside's Times and Software "Engineering" of Our Times. *IEEE Annals of the History of Computing*, 19(4):5–17.

[Bacławski et al., 1998] Bacławski, K., DeLoach, S. A., Kokar, M., e Smith, J. (1998). UML Formalization: A Position Paper. In *Thirteenth Annual Conference on Object-Oriented Programming, Languages and Applications (OOPSLA'98)*, Vancouver, Canadá.

[Barker, 1990] Barker, R. (1990). *CASE Method: Tasks and Deliverables*. Oracle.

[Barringer et al., 1997] Barringer, H., Fellows, D., Gough, G., e Williams, A. (1997). Abstract Modelling of Asynchronous Micropipeline Systems using Rainbow. In Kloos, C. D. e Cerny, E., editores, *Proceedings of the XIII IFIP Conference on Computer Hardware Description Languages and Their Applications (CHDL'97)*, pp. 285–304, Toledo, Espanha. Chapman & Hall.

[Barroca, 1992] Barroca, L. (1992). Using Real Time Logic to Prove Properties about Timed Statecharts and Transition Systems. In *Second International Workshop on Responsive Computer Systems*, Japão.

[Beck e Cunningham, 1989] Beck, K. e Cunningham, W. (1989). A Laboratory For Teaching Object-Oriented Thinking. *Sigplan Notices — OOPSLA'89*, 24(10):1–6. [c2.com/doc/oopsla89/paper.html].

[Benveniste e Berry, 1991] Benveniste, A. e Berry, G. (1991). The Synchronous Approach to Reactive and Real Time Systems. *Proceedings of the IEEE*, 79(9):1270–82.

[Benzakki e Djafri, 1997] Benzakki, J. e Djafri, B. (1997). Object Oriented Extensions to VHDL, The LaMI Proposal. In Kloos, C. D. e Cerny, E., editores, *Proceedings of the XIII IFIP Conference on Computer Hardware Description Languages and Their Applications (CHDL'97)*, pp. 334–47, Toledo, Espanha. Chapman & Hall.

[Bergé et al., 1996] Bergé, J. M., Levia, O., e Rouillard, J., editores (1996). *Object-Oriented Modeling*, volume 7 de *Current Issues in Electronic Modeling*. Kluwer Academic Publishers. ISBN 0-7923-9688-X.

[Bergé et al., 1997] Bergé, J. M., Levia, O., e Rouillard, J., editores (1997). *Hardware/Sofware Co-Design and Co-Verification*, volume 8 de *Current Issues in Electronic Modeling*. Kluwer Academic Publishers. ISBN 0-7923-9689-8.

[Bergner et al., 1998] Bergner, K., Rausch, A., e Sihling, M. (1998). A Critical Look upon UML 1.0. In Schader, M. e Korthaus, A., editores, *The Unified Modeling Language: Technical Aspects and Applications*, pp. 79–92, Heidelberg, Alemanha. Physica-Verlag.

[Boar, 1984] Boar, B. H. (1984). *Application Prototyping: A Requirements Definition Strategy for the 80's*. John Wiley & Sons. ISBN 0-471-89317-X.

[Boehm, 1988] Boehm, B. W. (1988). A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72.

[Booch, 1986] Booch, G. (1986). Object-Oriented Development. *IEEE Transactions on Software Engineering*, SE-12(2):211–21.

[Booch, 1991] Booch, G. (1991). *Object-Oriented Design with Applications*. Benjamin/Cummings. ISBN 0-8053-0091-0.

[Booch, 1994] Booch, G. (1994). *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, 2^a edição. ISBN 0-8053-5340-2.

[Booch, 1996] Booch, G. (1996). *Best of Booch: Designing Strategies for Object Technology*. SIGS. ISBN 1-884842-71-2.

[Booch et al., 1999] Booch, G., Rumbaugh, J., e Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Object Technology. Addison-Wesley. ISBN 0-201-57168-4.

[Bowen e Hinchey, 1995] Bowen, J. P. e Hinchey, M. G. (1995). The Commandments of Formal Methods. *IEEE Computer*, 28(4):56–63.

[Buchenrieder e Rozenblit, 1995] Buchenrieder, K. e Rozenblit, J. W. (1995). *Codesign — Computer-Aided Software/Hardware Engineering*, capítulo "Codesign: An Overview", pp. 1–15. IEEE Press.

[Budd, 1997] Budd, T. (1997). *An Introduction to Object-Oriented Programming.* Addison-Wesley, 2ª edição. ISBN 0-201-82419-1.

[Budgen, 1994] Budgen, D. (1994). *Software Design.* Addison-Wesley. ISBN 0-201-54403-2.

[Cabanis et al., 1996] Cabanis, D., Medhat, S., e Weavers, N. (1996). Object-Oriented Extensions to VHDL: The Classification Orientation. In *VHDL User Forum Europe — SIG-VHDL Spring'96 Working Conference*, pp. 9–20, Dresden, Alemanha. Shaker Verlag.

[Calvez, 1993] Calvez, J. P. (1993). *Embedded Real-Time Systems: A Specification and Design Methodology.* Software Engineering Practice. John Wiley & Sons. ISBN 0-471-93563-8.

[Calvez, 1996] Calvez, J. P. (1996). *High-Level System Modeling: and Design Methodologies*, volume 4 de *Current Issues in Electronic Modeling*, capítulo "A System Specification Model and Method", pp. 1–54. Kluwer Academic Publishers.

[Camposano e Wilberg, 1996] Camposano, R. e Wilberg, J. (1996). Embedded System Design. *Design Automation for Embedded Systems*, 1(1/2):5–50.

[Chatzoglou e Macaulay, 1995] Chatzoglou, P. D. e Macaulay, L. A. (1995). Requirements Capture and Analysis: the Project Manager's Dillema. *International Journal of Computer Applications in Technology*, 8(3/4):190–202.

[Clare, 1973] Clare, C. R. (1973). *Design Logic Systems Using State Machines.* McGraw-Hill.

[Coad e Yourdon, 1991] Coad, P. e Yourdon, E. (1991). *Object-Oriented Analysis.* Yourdon Press, 2ª edição. ISBN 0-13-629981-4.

[Coleman et al., 1994] Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., e Jeremaes, P. (1994). *Object-Oriented Development: The Fusion Method.* Prentice-Hall International. ISBN 0-13-101040-9.

[Cook, 1991] Cook, R. (1991). Embedded Systems in Control. *Byte*, 16(6):153–60.

[Cox e Novobilski, 1991] Cox, B. J. e Novobilski, A. J. (1991). *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley. ISBN 12667792.

[Davies e Layzell, 1993] Davies, C. G. e Layzell, P. J. (1993). *The Jackson Approach to System Development: an introduction.* Chartwell Bratt. ISBN 0-86238-321-8.

[Davis, 1994] Davis, A. M. (1994). *Software Requirements: Objects, Functions and States.* Prentice-Hall International, 2ª edição. ISBN 0-13-805763-X.

[de la Puente et al., 1998] de la Puente, J. A., León, G., Alonso, A., Dueñas, J. C., de Miguel, M. A., e Álvaro Redón (1998). Advanced Software Engineering Techniques for Real-Time Systems. In *9th IFAC Symposium on Information Control in Manufacturing (INCOM'98)*, volume 1, pp. 45–50, Nancy e Metz, França.

[De Micheli, 1994] De Micheli, G. (1994). *Synthesis and Optimization of Digital Circuits.* Electrical and Computer Engineering. McGraw-Hill. ISBN 0-07-016333-2.

[Delgado Kloos e Breuer, 1995] Delgado Kloos, C. e Breuer, P. T., editores (1995). *Formal Semantics for VHDL.* Engineering and Computer Science. Kluwer Academic Publishers. ISBN 0-7923-9552-2.

[DeMarco, 1979] DeMarco, T. (1979). *Structural Analysis and System Specification.* Prentice-Hall. ISBN 0-13-854380-1.

[Dias et al., 1996] Dias, O. P., Calha, M., Teixeira, I. C., e Teixeira, J. P. (1996). High-Level Test Specification for SFT using Object-Oriented Modeling Techniques. In *IEEE European Test Workshop (ETW'96)*, pp. 135–9, Montpellier, França.

[Douglass, 1997] Douglass, B. P. (1997). Designing Real-Time Systems With The Unified Modeling Language. *Electronic Design*, pp. 132–42.

[Douglass, 1998] Douglass, B. P. (1998). *Real-Time UML: Developing Efficient Objects for Embedded Systems.* Object Technology. Addison-Wesley. ISBN 0-201-32579-9.

[Douglass, 1999] Douglass, B. P. (1999). UML Statecharts. Relatório técnico, I-Logix, Andover, MA, E.U.A. `[www.ilogix.com/papers]`.

[Douglass et al., 1998] Douglass, B. P., Harel, D., e Trakhtenbrot, M. (1998). Statecharts in Use: Structured Analysis and Object-Orientation. In Vaandrager, F. e Rozenberg, G., editores, *Lectures on Embedded Systems*, volume 1494 de *Lecture Notes in Computer Science*, pp. 386–94. Springer-Verlag.

[Dunlop, 1994] Dunlop, D. D. (1994). Object-Oriented Extensions to VHDL. In *Proc. VIUF Fall 1994 Conference*, pp. 5.1–5.9.

[Ecker e Mrva, 1996] Ecker, W. e Mrva, M. (1996). *Object-Oriented Modeling*, volume 7 de *Current Issues in Electronic Modeling*, capítulo "Object Orientation: Modeling and Design Paradigms for the Year 2000?", pp. 1–14. Kluwer Academic Publishers.

[Embley et al., 1992] Embley, D. W., Kurtz, B. D., e Woodfield, S. N. (1992). *Object-Oriented Systems Analysis: A Model-Driven Approach.* Yourdon Press. ISBN 0-13-629973-3.

[Enciclopédia Luso-Brasileira de Cultura, 1973] Enciclopédia Luso-Brasileira de Cultura (1973). Verbo Editora, Lisboa, Portugal.

[Esteves et al., 1997] Esteves, A. J., Fernandes, J. M., e Proença, A. J. (1997). *Embedded System Applications*, capítulo "EDgAR: A Platform for Hardware/Software Codesign", pp. 19–32. Kluwer Academic Publishers. ISBN 0-7923-9947-1. `[www.di.uminho.pt/~miguel/PUBLI/atw96.ps]`.

[Eva, 1992] Eva, M. (1992). *SSADM version 4: A User's Guide.* Software Engineering. McGraw-Hill. ISBN 0-07-707409-2.

[Evans et al., 1998] Evans, A., Bruel, J.-M., France, R., Lano, K., e Rumpe, B. (1998). Making UML Precise. In *Thirteenth Annual Conference on Object-Oriented Programming, Languages and Applications (OOPSLA'98)*, Vancouver, Canadá.

[Fernandes, 1994] Fernandes, J. M. (1994). Redes de Petri e VHDL na Especificação de Controladores Paralelos. Tese de Mestrado, Dep. Informática, Universidade do Minho, Braga, Portugal. `[www.di.uminho.pt/~miguel/PUBLI/mestrado.ps]`.

[Fernandes et al., 1997] Fernandes, J. M., Adamski, M., e Proença, A. J. (1997). VHDL Generation from Hierarchical Petri Net Specifications of Parallel Controller. *IEE Proceedings: Computers and Digital Techniques*, 144(2):127–37. `[www.di.uminho.pt/~miguel/PUBLI/iee-cdt97.ps]`.

[Fernandes e Machado, 1997] Fernandes, J. M. e Machado, R. J. (1997). Projecto de Hardware Digital Orientado por Objectos. *Anais da Engenharia e Tecnologia Electrotécnica*, II(5):5–8. [www.di.uminho.pt/~miguel/PUBLI/aeteDez97.doc].

[Fernandes et al., 1999] Fernandes, J. M., Machado, R. J., e Santos, H. D. (1999). A UML-based Approach for Modeling Industrial Control Applications. In *2nd International Conference on the Unified Modeling Language (UML'99)*, Fort Collins, CO, E.U.A. Apresentado como poster. [www.di.uminho.pt/~miguel/PUBLI/uml99.ps].

[Fernandes et al., 1995] Fernandes, J. M., Pina, A. M., e Proença, A. J. (1995). Simulação e Síntese de Controladores Paralelos baseados em Redes de Petri. In *VII Simpósio Brasileiro de Arquitetura de Computadores — Processamento de Alto Desempenho (SBAC-PAD'95)*, pp. 481–92, Canela, Brasil. [www.di.uminho.pt/~miguel/PUBLI/sbacpad95.ps].

[Fernandes e Proença, 1994] Fernandes, J. M. e Proença, A. J. (1994). Redes de Petri na Especificação e Validação de Controladores Paralelos. In *1o. Encontro Nacional do Colégio de Engenharia Electrotécnica*, pp. 113–8, Ordem dos Engenheiros, Lisboa, Portugal. [www.di.uminho.pt/~miguel/PUBLI/encee94.ps].

[Ferreira, 1999] Ferreira, T. S. V. (1999). SSIT — Sistema de Supervisão de Iluminação. Relatório técnico, Escola Superior de Tecnologia, Instituto Politécnico de Castelo Branco, Castelo Branco, Portugal. Trabalho final de curso, Eng. Informática — Tecnologias da Informação.

[Fischer, 1989] Fischer, G. (1989). Human-Computer Interation in Software: Lessons Learned, Challenges Ahead. *IEEE Software*, 21(1):44–52.

[Flanagan, 1996] Flanagan, D. (1996). *Java in a Nutshell*. O'Reilly & Associates, Inc. ISBN 1-56592-183-6.

[Fowler e Scott, 1997] Fowler, M. e Scott, K. (1997). *UML Distilled: Applying the Standard Object Modeling Language*. Object Technology. Addison-Wesley. ISBN 0-201-32563-2.

[Gajski et al., 1992] Gajski, D. D., Dutt, N. D., Wu, A. C.-H., e Lin, S. Y.-L. (1992). *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers. ISBN 0-7923-9194-2.

[Gajski et al., 1993] Gajski, D. D., Vahid, F., e Narayan, S. (1993). SpecCharts: A VHDL Front-End for Embedded Systems. Relatório Técnico TR-93-31, Dept. of Information and Computer Science, University of California, Irvine, E.U.A.

[Gajski et al., 1994] Gajski, D. D., Vahid, F., Narayan, S., e Gong, J. (1994). *Specification and Design of Embedded Systems*. Prentice-Hall. ISBN 0-13-150731-1.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., e Vlissides, J. M. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing. Addison-Wesley. ISBN 0-201-63361-2.

[Gane e Sarson, 1979] Gane, C. e Sarson, T. (1979). *Structured Systems Analysis: Tools and Techniques*. Prentice-Hall. ISBN 0-13-854547-2.

[Geisler et al., 1998] Geisler, R., Klar, M., e Mann, S. (1998). Precise UML Semantics Through Formal Metamodeling. In *Thirteenth Annual Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'98)*, Vancouver, Canadá.

[Ghezzi et al., 1991] Ghezzi, C., Jazayeri, M., e Mandrioli, D. (1991). *Fundamentals of Software Engineering.* Prentice-Hall. ISBN 0-13-818204-3.

[Gibbs, 1994] Gibbs, W. (1994). Software's Chronic Crisis. *Scientific American*, 271(3):72–81.

[Giese et al., 1999] Giese, H., Graf, J., e Wirtz, G. (1999). Closing the Gap between Object-Oriented Modeling of Structure and Behavior. In *2nd International Conference on the Unified Modeling Language (UML'99)*, pp. 534–49, Fort Collins, CO, E.U.A. Springer-Verlag.

[Glunz, 1991] Glunz, W. (1991). Extensions from VHDL to VHDL++. Relatório Técnico JESSI-AC/S2-WP1-T2.4-Q3, ZFE BT SE 61, Siemens AG.

[Goldberg e Robson, 1983] Goldberg, A. e Robson, D. (1983). *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley.

[Gomes, 1997] Gomes, L. F. (1997). *Redes de Petri Reactivas e Hierárquicas: Integração de Formalismos no Projecto de Sistemas Reactivos de Tempo-real.* Tese de Doutoramento, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa.

[Graham, 1991] Graham, I. (1991). *Object-Oriented Methods.* Addison-Wesley. ISBN 0-201-56521-8.

[Grande Enciclopédia Portuguesa e Brasileira, 1975] Grande Enciclopédia Portuguesa e Brasileira (1975). Editorial Enciclopédia, Lisboa, Portugal e Rio de Janeiro, Brasil.

[Gupta, 1997] Gupta, R. K. (1997). *Co-Synthesis of Hardware and Sofware for Digital Embedded Systems*, volume 329 de *Engineering and Computer Science.* Kluwer Academic Publishers.

[Guttag, 1993] Guttag, J. (1993). A Brief Introduction to CLU. *ACM SIGPLAN Notices*, 28(3):351–2.

[Harel, 1987] Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–74.

[Harel, 1992] Harel, D. (1992). Biting the Silver Bullet: Toward a Brighter Future for System Development. *IEEE Computer*, 25(1):8–20.

[Harel, 1999] Harel, D. (1999). On the Behavior of Complex Object-Oriented Systems. In *2nd International Conference on the Unified Modeling Language (UML'99)*, pp. 324–9, Fort Collins, CO, E.U.A. Springer-Verlag.

[Harel e Gery, 1997] Harel, D. e Gery, E. (1997). Executable Object Modeling with Statecharts. *IEEE Computer*, 30(7):31–42.

[Harel et al., 1990] Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, A., e Trakhtenbrot, M. (1990). STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16(4):403–14.

[Harel e Naamad, 1996] Harel, D. e Naamad, A. (1996). The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering Method*, 5(4):293–333.

[Hatley e Pirbhai, 1988] Hatley, D. J. e Pirbhai, I. A. (1988). *Strategies for Real-Time System Specification*. Dorset House. ISBN 0-932633-11-0.

[Herzberg, 1999] Herzberg, D. (1999). UML-RT as a Candidate for Modeling Embedded Real-Time Systems in the Telecommunication Domain. In *2nd International Conference on the Unified Modeling Language (UML'99)*, pp. 330–8, Fort Collins, CO, E.U.A. Springer-Verlag.

[Hoare, 1978] Hoare, C. A. R. (1978). Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–77.

[Hutt, 1994] Hutt, A. T. F., editor (1994). *Object Analysis and Design: Description of Methods*. John Wiley & Sons. ISBN 0-471-62366-0.

[IEEE, 1994] IEEE (1994). *IEEE Standard VHDL Language Reference Manual (ANSI/IEEE Std 1076-1993)*. IEEE Inc., New York, E.U.A. ISBN 1-55937-376-8.

[Jacobson et al., 1992] Jacobson, I., Christerson, M., Jonsson, P., e Övergaard, G. (1992). *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley. ISBN 0-201-54435-0.

[Kabous e Nebel, 1999] Kabous, L. e Nebel, W. (1999). Modeling Hard Real-Time Systems with UML: The OOHARTS Approach. In *2nd International Conference on the Unified Modeling Language (UML'99)*, pp. 339–55, Fort Collins, CO, E.U.A. Springer-Verlag.

[Kelton et al., 1998] Kelton, W. D., Sadowski, R. P., e Sadowski, D. A. (1998). *Simulation with Arena*. McGraw-Hill. ISBN 0-07-561259-3.

[Kit, 1995] Kit, E. (1995). *Software Testing in the Real World: Improving the Process*. Addison-Wesley. ISBN 0-201-87756-2.

[Kleinjohann et al., 1997] Kleinjohann, B., Tacken, J., e Tahedl, C. (1997). Towards a Complete Design Method for Embedded Systems Using Predicate/Transition-Nets. In Kloos, C. D. e Cerny, E., editores, *Proceedings of the XIII IFIP Conference on Computer Hardware Description Languages and Their Applications (CHDL'97)*, pp. 4–23, Toledo, Espanha. Chapman & Hall.

[Kumar et al., 1994] Kumar, S., Aylor, J. H., Johnson, B. W., e Wulf, W. A. (1994). Object-Oriented Techniques in Hardware Design. *IEEE Computer*, 27(6):64–70.

[Kumar et al., 1996a] Kumar, S., Aylor, J. H., Johnson, B. W., e Wulf, W. A. (1996). *Object-Oriented Modeling*, volume 7 de *Current Issues in Electronic Modeling*, capítulo "Object-Oriented Modeling of Hardware for Embedded Systems", pp. 15–37. Kluwer Academic Publishers.

[Kumar et al., 1996b] Kumar, S., Aylor, J. H., Johnson, B. W., e Wulf, W. A. (1996). *The Codesign of Embedded Systems: A Unified Hardware/Software Representation*. Kluwer Academic Publishers. ISBN 0-201-49834-0.

[Lanusse et al., 1998] Lanusse, A., Gérard, S., e Terrier, F. (1998). Real-Time Modeling with UML: The ACCORD Approach. In *Proceedings of the International Workshop on the Unified Modeling Language: Beyond the Notation (UML'98)*, Mulhouse, França.

[Lavagno et al., 1996] Lavagno, L., Sangiovanni-Vincentelli, A., e Hsieh, H. (1996). Models and Algorithms for Embedded System Synthesis and Validation. In De Micheli, G., editor, *Nato Advanced Study Institute*. Kluwer Academic Publisher.

[Law e Kelton, 1991] Law, A. M. e Kelton, W. D. (1991). *Simulation Modeling and Analysis*. McGraw-Hill, 2ª edição. ISBN 0-07-036698-5.

[Leblanc, 1996] Leblanc, P. (1996). *Object-Oriented Modeling*, volume 7 de *Current Issues in Electronic Modeling*, capítulo "Object-Oriented and Real-Time Techniques: Combined Use of OMT, SDL and MSC", pp. 39–55. Kluwer Academic Publishers.

[Ledgard, 1981] Ledgard, H. (1981). *ADA: An Introduction*. Springer-Verlag. ISBN 0-387-90568-5.

[Lewis, 1992] Lewis, R. O. (1992). *Independent Verification and Validation: A Life Cycle Engineering Process for Quality Software*. John Wiley & Sons. ISBN 0-471-57011-7.

[Liskov, 1993] Liskov, B. (1993). A History of CLU. *ACM SIGPLAN Notices*, 28(3):133–147.

[Loucopoulos e Karakostas, 1995] Loucopoulos, P. e Karakostas, V. (1995). *Software Requirements Engineering*. McGraw-Hill International. ISBN 0-07-707843-8.

[Lyons, 1998] Lyons, A. (1998). UML for Real-Time Overview. Relatório técnico, ObjecTime Limited. `[www.objectime.com/otl/technical/umlrt_overview.pdf]`.

[Macaulay, 1996] Macaulay, L. A. (1996). *Requirements Engineering*. Applied Computing. Springer-Verlag, Londres, Reino Unido. ISBN 3-540-76006-9.

[Machado, 1996] Machado, R. J. (1996). Hierarquia em Redes de Petri Orientadas por Objectos na Especificação de Sistemas Digitais. Tese de Mestrado, Dep. Informática, Universidade do Minho, Braga, Portugal.

[Machado et al., 1997a] Machado, R. J., Fernandes, J. M., e Proença, A. J. (1997). Redes de Petri e VHDL na Prototipagem Rápida de Sistemas Digitais. *Anais da Engenharia e Tecnologia Electrotécnica*, II(4):1–4. `[www.di.uminho.pt/~miguel/PUBLI/aeteJul97.doc]`.

[Machado et al., 1997b] Machado, R. J., Fernandes, J. M., e Proença, A. J. (1997). SOFHIA: A CAD Environment to Design Digital Control Systems. In Kloos, C. D. e Cerny, E., editores, *Proceedings of the XIII IFIP Conference on Computer Hardware Description Languages and Their Applications (CHDL'97)*, pp. 86–8, Toledo, Espanha. Chapman & Hall. `[www.di.uminho.pt/~miguel/PUBLI/chdl97.ps]`.

[Machado et al., 1997c] Machado, R. J., Fernandes, J. M., e Proença, A. J. (1997). Specification of Industrial Digital Controllers with Object-Oriented Petri Nets. In *IEEE International Symposium on Industrial Electronics (ISIE'97)*, volume 1, pp. 78–83, Guimarães, Portugal. `[www.di.uminho.pt/~miguel/PUBLI/isie97.ps]`.

[Machado et al., 1998a] Machado, R. J., Fernandes, J. M., e Proença, A. J. (1998). An Object-Oriented Model for Rapid Prototyping of Data Path/Control Systems — A Case Study. In *9th IFAC Symposium on Information Control in Manufacturing (INCOM'98)*, volume 2, pp. 269–74, Nancy e Metz, França. [www.di.uminho.pt/~miguel/PUBLI/incom98.ps].

[Machado et al., 1998b] Machado, R. J., Fernandes, J. M., e Proença, A. J. (1998). Hierarchical Mechanisms for High-level Modelling and Simulation of Digital Systems. In *5th IEEE International Conference on Electronics, Circuits and Systems (ICECS'98)*, volume 3, pp. 229–32, Lisboa, Portugal. [www.di.uminho.pt/~miguel/PUBLI/icecs98.doc].

[Maffezzoni et al., 1998] Maffezzoni, C., Ferrarini, L., e Carpanzo, E. (1998). Object-Oriented Models for Advanced Automation Engineering. In *9th IFAC Symposium on Information Control in Manufacturing (INCOM'98)*, volume 1, pp. 21–31, Nancy e Metz, França.

[Mariatos et al., 1996] Mariatos, E. P., Birbas, A. N., Birbas, M. K., Karathanasis, I., Jadoul, M., Verschaeve, K., Roux, J.-L., e Sinclair, D. (1996). *Object-Oriented Modeling*, volume 7 de *Current Issues in Electronic Modeling*, capítulo "Integrated System Design with an Object-Oriented Methodology", pp. 57–75. Kluwer Academic Publishers.

[Martin e Odell, 1992] Martin, J. e Odell, J. J. (1992). *Object-Oriented Analysis and Design*. Prentice-Hall.

[Martins, 1998] Martins, F. M. (1998). Camila: Uma Abordagem Moderna e Rigorosa para a Engenharia Informática. *Anais da Engenharia e Tecnologia Electrotécnica*, II(5):1–4.

[McLaughin e Moore, 1998] McLaughin, M. J. e Moore, A. (1998). Real-Time Extensions to UML. *Dr. Dobb's Journal*, (292):82–93.

[Metz et al., 1999] Metz, P., O'Brien, J., e Weber, W. (1999). Code Generation Concepts for Statechart Diagrams of the UML v1.1. In *Object Technology Group Conference*, Viena, Áustria.

[Meyer, 1988] Meyer, B. (1988). *Object-Oriented Software Construction*. Computer Science. Prentice-Hall. ISBN 0-13-629031-0.

[Miller, 1957] Miller, G. A. (1957). The Magical Number 7 Plus or Minus 2: Some Limitations on Our Capacity for Processing Information. *Psychological Review*, (63):81–97.

[Mills, 1988] Mills, H. D. (1988). Stepwise Refinement and Verification in Box-Structured Systems. *IEEE Computer*, 21(6).

[Monarchi, 1994] Monarchi, D. (1994). Methodology Standards: Help or Hindrance? (Panel 4). In *Ninth Annual Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'94), ACM Sigplan Notices*, volume 29, pp. 223–8.

[Moore, 1965] Moore, G. E. (1965). Cramming More Components Onto Integrated Circuits. *Electronics*, 38(8):114–7.

[Morris et al., 1996] Morris, D., Evans, G., Green, P., e Theaker, C. (1996). *Object-Oriented Computer Systems Engineering*. Applied Computing. Springer-Verlag, Londres, Reino Unido. ISBN 3-540-76020-2.

[Mota, 1999] Mota, M. (1999). A Engenharia no Século XXI. Oração de Sapiência, Escola de Engenharia, Universidade do Minho, Braga, Portugal.

[Müller e Rammig, 1989] Müller, W. e Rammig, F. (1989). ODICE: Object-Oriented Hardware Description in CAD Environment. In *Proceedings of the Ninth IFIP International Symposium on Computer Hardware Description Language and their Applications*, pp. 19–34. North Holland.

[Murata, 1989] Murata, T. (1989). Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–80.

[Myers, 1978] Myers, G. (1978). *Composite/Structured Design*. Van Nostrand Reinhold, New York, E.U.A.

[Naughton, 1996] Naughton, P. (1996). *The Java Handbook*. Osborne McGraw-Hill. ISBN 0-07-882199-1.

[Nebel e Schumacher, 1996] Nebel, W. e Schumacher, G. (1996). Object-Oriented Hardware Modelling – Where to Apply and What are the Objects? In *Proceedings of Euro-DAC'96 with Euro-VHDL'96*, pp. 428–33, Genève, Suíça. IEEE Computer Society Press.

[Oblog, 1997a] Oblog (1997). Oblog Language Guide. Relatório técnico, Oblog Softwarec S.A.

[Oblog, 1997b] Oblog (1997). Oblog Language Semantic Aspects Guide. Relatório técnico, Oblog Software S.A.

[Oblog, 1997c] Oblog (1997). Oblog Language Types and Expressions Guide. Relatório técnico, Oblog Software S.A.

[Oliveira, 1995] Oliveira, J. N. (1995). Especificação e Desenvolvimento Formal de Programas. Relatório técnico, Dep. Informática, Universidade do Minho, Braga, Portugal.

[Övergaard, 1998] Övergaard, G. (1998). A Formal Approach to Relationships in the Unified Modeling Language. In Broy, M., Coleman, D., Maibaum, T. S. E., e Rumpe, B., editores, *Proceedings of the Workshop on Precise Semantics for Software Modeling Techniques (PSMT'98)*, pp. 91–108, Munique, Alemanha. Technische Universität München, TUM-I9803.

[Övergaard e Palmkvist, 1998] Övergaard, G. e Palmkvist, K. (1998). A Formal Approach to Use Cases and Their Relationships. In *Proceedings of the International Workshop on the Unified Modeling Language: Beyond the Notation (UML'98)*, Mulhouse, França.

[Padilla, 1993] Padilla, A. J. G. (1993). *Sistemas Digitais*. McGraw-Hill, Lisboa, Portugal. ISBN 972-9241-43-0.

[Page-Jones, 1988] Page-Jones, M. (1988). *Practical Guide to Structured Systems Design*. Yourdon Press, 2^a edição. ISBN 0-13-690769-5.

[Pascoe, 1986] Pascoe, G. A. (1986). Elements of Object-Oriented Programming. *Byte*, 11(8):139–44.

[Pereira e Paredes, 2000] Pereira, F. e Paredes, H. (2000). Geração de Scripts: Oblog para VHDL. Relatório técnico, Dep. Informática, Universidade do Minho, Braga, Portugal. Trabalho de Projecto de Opção III, Eng. de Sistemas e Informática.

[Perry, 1992] Perry, D. (1992). Applying Object Oriented Techniques to VHDL. In *Proceedings of the VIUF Spring 1992 Conference*, pp. 217–24.

[Pressman, 1997] Pressman, R. S. (1997). *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 4ª edição. ISBN 0-07-052182-4.

[Putzke-Raming et al., 1997] Putzke-Raming, W., Radetzki, M., e Nebel, W. (1997). Objective VHDL: Requirements Collection and Design Objectives for object-oriented extensions to VHDL. Relatório técnico, OFFIS, Oldenburg, Alemanha. `[eis.informatik.uni-oldenburg.de/research/request_public/requirements.rtf.gz]`.

[Robinson, 1992] Robinson, P. J. (1992). *Hierarchical Object-Oriented Design*. The Object-Oriented Series. Prentice-Hall. ISBN 0-13-390816-X.

[Robson, 1981] Robson, D. (1981). Object-Oriented Software Systems. *Byte*, 6(8):74–86.

[Rumbaugh et al., 1991] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., e Lorensen, W. (1991). *Object-Oriented Modeling and Design*. Prentice-Hall International. ISBN 0-13-630054-5.

[Rumbaugh et al., 1999] Rumbaugh, J., Jacobson, I., e Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Object Technology. Addison-Wesley. ISBN 0-201-30998-X.

[Schneider e Winters, 1998] Schneider, G. e Winters, J. P. (1998). *Applying Use Cases: A Pratical Guide*. Object Technology. Addison-Wesley. ISBN 0-201-30981-5.

[Schumacher e Nebel, 1995a] Schumacher, G. e Nebel, W. (1995). *High-Level System Modeling: Specification Languages*, volume 3 de *Current Issues in Electronic Modeling*, capítulo "Survey on Object-Oriented Languages for Hardware Design Methodologies. Kluwer Academic Publishers.

[Schumacher e Nebel, 1995b] Schumacher, G. e Nebel, W. (1995). Inheritance Concept for Signals in Object-Oriented Extensions to VHDL. In *Proceedings of Euro-DAC'95 with Euro-VHDL'95*, pp. 428–35, Brighton, Reino Unido. IEEE Computer Society Press.

[Schumacher et al., 1996] Schumacher, G., Nebel, W., Putzke, W., e Wilmes, M. (1996). Applying Object-Oriented Techniques to Hardware Modelling – A Case Study. In *VHDL User Forum Europe — SIG-VHDL Spring'96 Working Conference*, pp. 1–8, Dresden, Alemanha. Shaker Verlag.

[Selic et al., 1994] Selic, B., Gullekson, G., e Ward, P. T. (1994). *Real-Time Object-Oriented Modeling*. John Wiley & Sons. ISBN 0-471-59917-4.

[Selic e Rumbaugh, 1998] Selic, B. e Rumbaugh, J. (1998). Using UML for Modeling Complex Real-Time Systems. Relatório técnico, ObjecTime Limited & Rational Software. `[www.objectime.com/otl/technical/umlrt.pdf]`.

[Shlaer e Mellor, 1992] Shlaer, S. e Mellor, S. J. (1992). *Object-Lifecycles — Modeling the World in States*. Prentice-Hall. ISBN 0-13-629940-7.

[Sigfried, 1996] Sigfried, S. (1996). *Understanding Object-Oriented Software Engineering*. IEEE Press. ISBN 0-7803-1082-9.

[Sklyarov et al., 1998] Sklyarov, V., Rocha, A. A., e Ferrari, A. B. (1998). *Advanced Techniques for Embedded Systems Design and Test*, capítulo "Synthesis of Reconfigurable Control Devices Based on Object-Oriented Specifications", pp. 151–77. Kluwer Academic Publishers. ISBN 0-7923-8128-9.

[Smith e Gross, 1986] Smith, C. U. e Gross, R. R. (1986). Technology Transfer Between VLSI Design and Software Engineering: CAD Tools and Design Methodologies. *Proceedings of the IEEE*, 74(6):875–85.

[Smith e Tockey, 1988] Smith, M. M. e Tockey, S. R. (1988). An Integrated Approach to Software Requirements Definition Using Objects. In *Tenth Structured Development Forum*, San Francisco, CA, E.U.A.

[Stevens et al., 1998] Stevens, R., Brook, P., Jackson, K., e Arnold, S. (1998). *Systems Engineering: Coping with Complexity*. Prentice-Hall Europe. ISBN 0-13-095085-8.

[Stroustrup, 1987] Stroustrup, B. (1987). What is "Object-Oriented Programming?". In *European Conference on Object-Oriented Programming (ECOOP'87)*.

[Stroustrup, 1991] Stroustrup, B. (1991). *The C++ Programming Language*. Addison-Wesley, 2^a edição. ISBN 0-201-53992-6.

[Sully, 1993] Sully, P. (1993). *Modelling the World With Objects*. Prentice-Hall. ISBN 0-13-587791-1.

[Swamy et al., 1995] Swamy, S., Molin, A., e Covnot, B. (1995). OO-VHDL: Object-Oriented Extensions to VHDL. *IEEE Computer*, 28(10):18–26.

[Szyperski, 1998] Szyperski, C. (1998). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley. ISBN 0-201-17888-5.

[Taenzer, 1989] Taenzer, D. H. (1989). The yo-yo Problem. *Journal of Object-Oriented Programming*, 2(3):30–5.

[Tanenbaum, 1976] Tanenbaum, A. S. (1976). *Structured Computer Organization*. Prentice-Hall, 1^a edição. ISBN 0-13-854497-2.

[Tanenbaum, 1990] Tanenbaum, A. S. (1990). *Structured Computer Organization*. Prentice-Hall, 3^a edição. ISBN 0-13-852872-1.

[Tanenbaum, 1999] Tanenbaum, A. S. (1999). *Structured Computer Organization*. Prentice-Hall, 4^a edição. ISBN 0-13-020435-8.

[Team BP-UM, 1999a] Team BP-UM (1999). Diagnóstico e Optimização do Sistema de Controlo das Linhas HIDRO — Documento de Requisitos. Relatório técnico, Blaupunkt & Universidade do Minho, Braga, Portugal.

[Team BP-UM, 1999b] Team BP-UM (1999). Diagnóstico e Optimização do Sistema de Controlo das Linhas HIDRO — Documento do Sistema Legado. Relatório técnico, Blaupunkt & Universidade do Minho, Braga, Portugal.

[Teorey, 1990] Teorey, T. J. (1990). *Database Modeling and Design: The Entity Relationship Approach*. Morgan Kaufman Publisher, San Metro, CA, E.U.A.

[Thomé, 1993] Thomé, B., editor (1993). *Systems Engineering: Principles and Practice of Computer-Based Systems Engineering*. John Wiley & Sons. ISBN 0-471-93552-2.

[Thomas, 1989] Thomas, D. (1989). What is an Object? *Byte*, 14(3):231–40.

[Thomas et al., 1993] Thomas, D. E., Adams, J. K., e Schmit, H. (1993). A Model and Methodology for Hardware-Software Codesign. *IEEE Design & Test of Computers*, 10(3):6–15.

[Tichy, 1998] Tichy, W. F. (1998). Should Computer Scientists Experiment More? *IEEE Computer*, 31(5):32–40.

[Tockey et al., 1990] Tockey, S. R., Hoza, B., e Cohen, S. (1990). Object-Oriented Analysis: Building on the Structured Techniques. In *Software Improvement Conference*, Washington, DC, E.U.A. Educational Foundation of the Data Processing Management Association.

[Unified Modeling Language, 1997a] Unified Modeling Language (1997). UML Metamodel. Relatório técnico, Rational Software.

[Unified Modeling Language, 1997b] Unified Modeling Language (1997). UML Semantics. Relatório técnico, Rational Software.

[Unified Modeling Language, 1997c] Unified Modeling Language (1997). UML Summary. Relatório técnico, Rational Software.

[van Emde Boas, 1998] van Emde Boas, P. (1998). Formalizing UML; Mission Impossible? In *Thirteenth Annual Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'98)*, Vancouver, Canadá.

[Verschueren, 1992] Verschueren, A. C. (1992). *An Object-Oriented Modelling Technique for Analysis and Design of Complex (Real-Time) Systems*. Tese de Doutoramento, Technische Universiteit Eindhoven. ISBN 90-9005046-9.

[Ward e Mellor, 1985] Ward, P. e Mellor, S. (1985). *Structured Development for Real-Time Systems*. Yourdon Press.

[Weber e Metz, 1998] Weber, W. e Metz, P. (1998). Reuse of Models and Diagrams of the UML and Implementation Concepts Regarding Dynamic Modeling. In Schader, M. e Korthaus, A., editores, *The Unified Modeling Language: Technical Aspects and Applications*, pp. 190–203, Heidelberg, Alemanha. Physica-Verlag.

[Wegner, 1990] Wegner, P. (1990). Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger*, 1(1):7–87.

[Whytock, 1993] Whytock, S. (1993). *Systems Engineering: Principles and Practice of Computer-Based Systems Engineering*, capítulo "The Development Life-Cycle", pp. 81–96. Software Based Systems. John Wiley & Sons.

[Williams, 1992] Williams, T. (1992). Object-Oriented Methods Transform Real-Time Programming. *Computer Design*, 31(9):100–18.

[Wilson, 1987] Wilson, R. (1987). Embedded Systems Manipulate Distributed Tasks. *Computer Design*, 26(16):49–61.

[Wilson, 1989] Wilson, R. (1989). Higher Speeds push Embedded Systems to Multiprocessing. *Computer Design*, 28(13):72–83.

[Wirfs-Brock e Wilkerson, 1989] Wirfs-Brock, R. e Wilkerson, B. (1989). Object-Oriented Design: A Responsability-Driven Approach. *Sigplan Notices — OOPSLA'89*, 24(10):71–6.

[Wirth, 1976] Wirth, N. (1976). *Algorithms + Data Structures = Programs*. Prentice-Hall.

[Yen e Wolf, 1996] Yen, T.-Y. e Wolf, W. (1996). *Hardware/Sofware Co-Synthesis of Distributed Embedded Systems*. Kluwer Academic Publishers. ISBN 0-7923-9797-5.

[Yourdon, 1988] Yourdon, E. (1988). *Managing the System Life Cycle*. Prentice-Hall, 2^a edição. ISBN 0-13-547530-9.

[Yourdon, 1989] Yourdon, E. (1989). *Modern Structured Analysis*. Prentice-Hall. ISBN 0-13-598624-9.

[Yourdon e Constantine, 1979] Yourdon, E. e Constantine, L. (1979). *Structured Design*. Prentice-Hall.

[Zave, 1982] Zave, P. (1982). An Operational Approach to Requirements Specification for Embedded Systems. *IEEE Transactions on Software Engineering*, SE-8(3):250–69.

[Zave, 1984] Zave, P. (1984). The Operational versus the Conventional Approach to Software Development. *Communications of the ACM*, 27(2):104–18.

Índice