Especificação Formal, Prototipagem e Cálculo de Programas — Um Caso na Indústria

Comunicação

Inforum'94 — Controlo de Qualidade em Software
7–8 Outubro
Braga

J. Nuno Oliveira
Universidade do Minho / INESC (Grupo 2361)
Braga

email: jno@di.uminho.pt

Sumário

- Após mais de duas décadas de intensa investigação fundamental, os **métodos formais de programação** aparecem neste final de século como uma promissora disciplina de desenvolvimento de 'software' capaz de
  - introduzir **rigor científico** no acto de programar, convertendo em *engenharia* uma actividade tradicionalmente viciada pelo amadorismo;
  - criar uma base científica para a hoje tão necessária **certificação** do 'software'.
- Após uma breve resenha da evolução histórica dos métodos formais de programação, esta comunicação descreve um caso de aplicação industrial desses métodos, ilustrando de forma clara o **ciclo de vida** de uma aplicação quando abordada e desenvolvida sob uma perspectiva formal.
- A abordagem subjacente — a abordagem **CAMILA** — está desde há alguns anos em desenvolvimento no INESC/ Universidade do Minho.
- Principal objectivo de **CAMILA**: a promoção do emprego de métodos formais no desenvolvimento **industrial** de 'software'.

Caso na Indústria

Ficha técnica:

- Contrato de I&D *Eureka 379* — **SOUR**
- Parceiros: *Systena* (Grupo Olivetti - Roma), *Sss* (Grupo Olivetti - Bari), *Olivetti Ricerca* (Grupo Olivetti - Bari), *Inesc* (Grupo 2361 - Braga).
- Período de execução: 1989–94.
- Orçamento total: $\simeq$ 16 M Liras.

Objectivos:

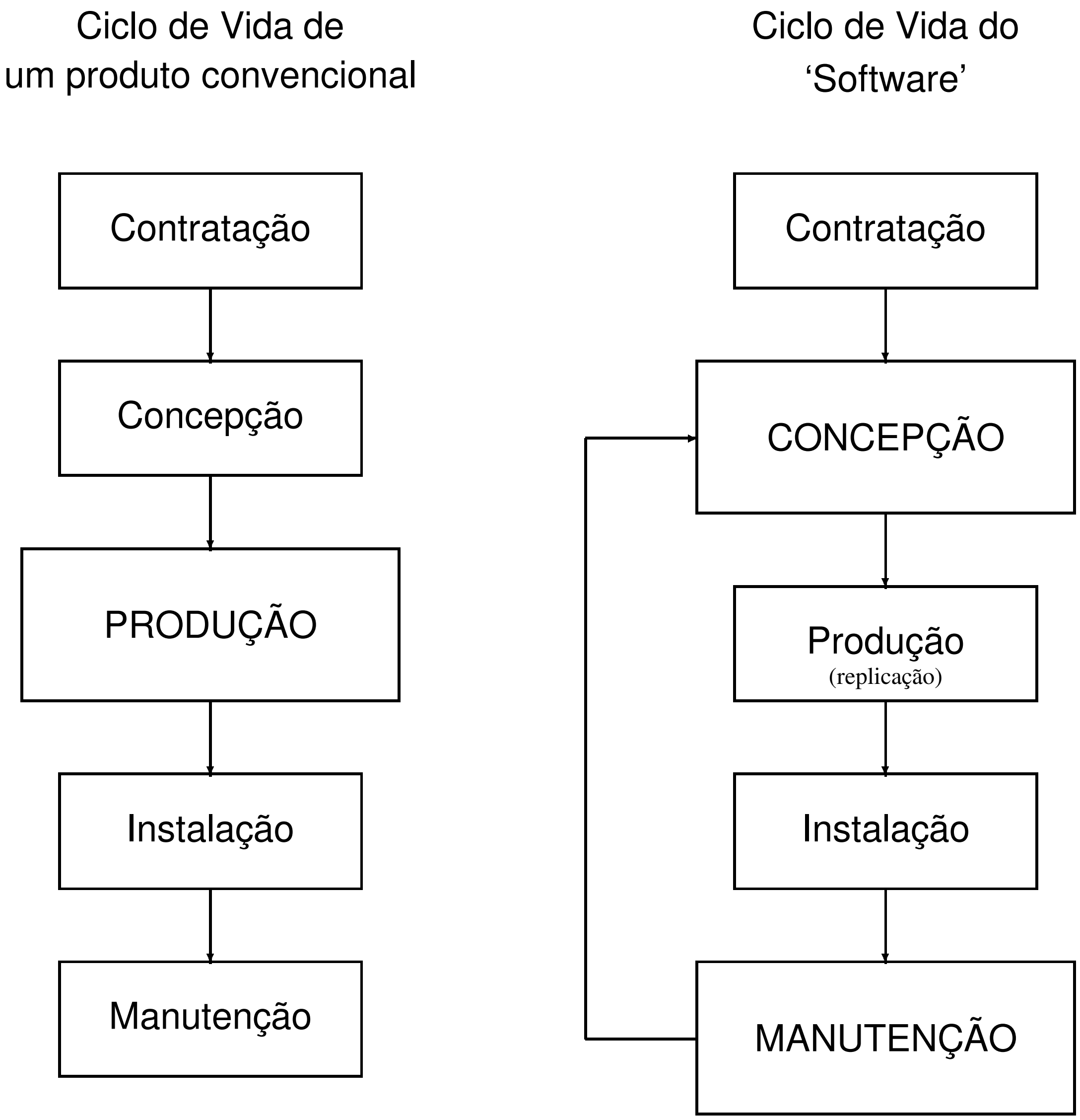
- Sistema: *Case* capaz de promover a re-utilização de 'software' (SOUR=<u>So</u>ftware <u>U</u>se and <u>R</u>euse).
- Ferramentas: caracterização, comparação, modificação e re-utilização de código e outra documentação técnica.
- Plataforma: *Osa/R*(epository) da Olivetti sob *Windows 3.1*.

'Workpackages' INESC:

- *Con*(ceptualizer);
- *Iqs* (Intelligent Query System);
- *CMod* (Comparator & Modifier);
- *RM* (Result Manager).

Enquadramento

- Controlo da qualidade do 'software' enquanto produto industrial.
- Ênfase na concepção:



cf. [*Vicente 94*].

- Binómio Especificação-Implementação:

$$\begin{array}{rcl} \text{especificação} & \boxed{\text{O quê}} & \text{(abstracção)} \\ & \uparrow & \\ \text{implementação} & \boxed{\text{Como}} & \text{(eficiência)} \end{array}$$

- Antinómio "Informal"/"Formal".
- 30 anos de "Crise de 'Software'"...

Um "Pecado Original" ?

O da Engenharia de 'Software' parece ter consistido em...

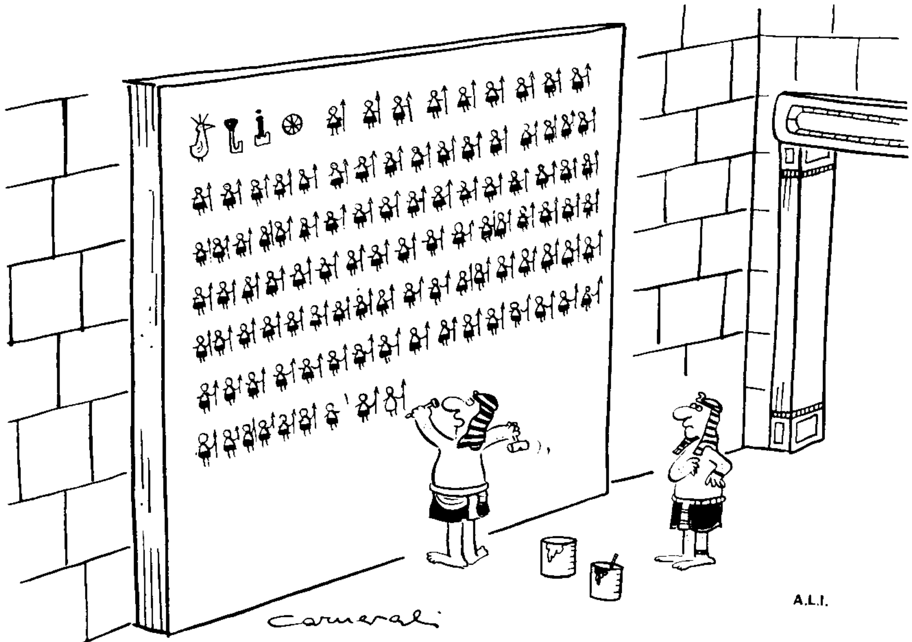
> ...atrever-se a ser "engenharia" sem ter uma *ciência* de suporte.

Sob o ponto de vista científico, o que faz falta é:

$$\begin{array}{cl} \boxed{\text{O quê}} & \text{(especificação)} \\ \uparrow & \\ \boxed{\boxed{\text{Porquê}}} & \textit{(justificação)} \\ \uparrow & \\ \boxed{\text{Como}} & \text{(implementação)} \end{array}$$

Requisitos

- uma “boa” notação...
- ... matematicamente respeitável (analogia: o **cálculo integral-diferencial** que suporta as engenharias clássicas).



Are you sure there isn't a simpler means of writing 'The Pharaoh had 10,000 soldiers?'

(reprodução de [*Jones 80*])

Abordagem Seguida

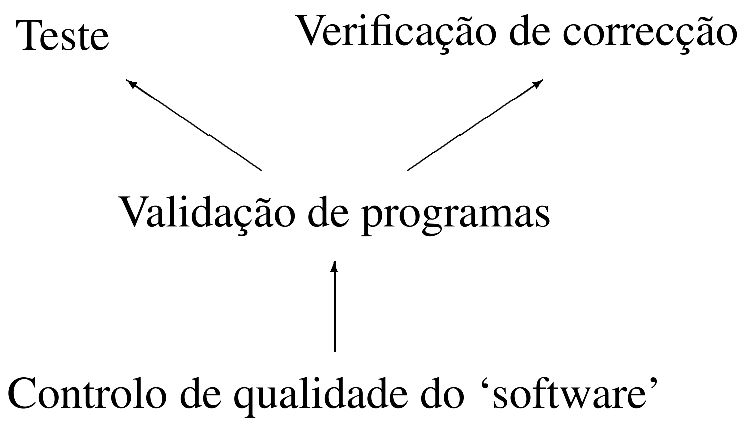
... Métodos formais para a *especificação e desenvolvimento científico* do ‘software’.

Contudo. . .

- . . . os métodos formais são quase tão “velhos” quanto a crise.
- . . . o que tem estado a falhar?

Perspectiva Histórica

Anos 60. . . 70. . .



Teste (vulg. ‘debug’)

- A primeira forma de controlo da qualidade de um programa que se conhece e, ainda hoje e em larga medida, a mais utilizada [1].
- Inconvenientes (conhecidos):
  - método anti-científico;

[1] Consiste em corrigir um programa por execuções sucessivas, num processo de “erro e nova tentativa” (típico dos seres irracionais).

- ineficácia:

  ... o teste de um programa apenas demonstra a existência *de erros, nunca a sua ausência. (E.W. Dijkstra)* [2]

- aplicação restrita;
- custos elevados [3];
- método irresponsável;
- má documentação (se alguma);
- agravamento dos custos da manutenção.

Verificação Formal

- Forma alternativa para controlo da qualidade dos programas que se demarca em relação ao *teste* ao pretender verificar a correcção de um programa **sem** que se precise de o executar.
- Recurso à Lógica e à Semântica Denotacional (a "novidade" dos anos 60).

$$\text{Dados} \left\{ \begin{array}{lcl} E & = & \text{especificação formal} \\ P & = & \text{programa candidato a implementação} \end{array} \right.$$

Verificação consiste em provar

$$[\![P]\!] \; \underline{\text{sat}} \; E$$

onde $[\![P]\!]$ = semântica formal de P.

Inconveniente: e se $wc(P) \rightarrow \infty \ldots$?

[2] Tem-se: teste revela erros $\Rightarrow$ programa tem erros, quer dizer, teste não revela erros $\not\Rightarrow$ programa não tem erros.

[3] Pessoal $\times$ tempo $+$ excessiva dependência de recursos computacionais.

Necessidade de uma alternativa:

$$\text{Validação} \begin{cases} \text{Teste} \\ \text{Verificação Formal} \begin{cases} \text{Estática ('a posteriori')} \\ ? \end{cases} \end{cases}$$

Verificação Construtiva

Alternativa que "explica" historicamente a **Programação Estruturada** dos anos 70...:

Por muito complicada que E possa ser, temos sempre

$$E = \varphi(E_1, \ldots, E_n)$$

Estratégia ($i = 1, n$):

- cada E_i é "sub-especificação" (especifica "uma parte" do problema original);
- conjecturar P_i tal que
  $$[\![P_i]\!] \ \underline{\mathsf{sat}}\ E_i$$
- procurar $\mathcal{E}$ que combine os n-subprogramas P_i
  $$\mathcal{E}(P_1, \ldots, P_n)$$
  tal que:
  $$\frac{[\![P_1]\!] \ \underline{\mathsf{sat}}\ E_1 \wedge \ldots \wedge [\![P_n]\!] \ \underline{\mathsf{sat}}\ E_n}{[\![\underbrace{\mathcal{E}(P_1, \ldots, P_n)}_{P}]\!] \ \underline{\mathsf{sat}}\ \underbrace{\varphi(E_1, \ldots, E_n)}_{E}} \quad (1)$$
- se $wc(P_i)$ ainda incomportável, repetir estratégia estruturalmente:
  - para demonstrar
    $$[\![P_i]\!] \ \underline{\mathsf{sat}}\ E_i$$

onde

$$E_i = \varphi_i(E_{i_1}, \ldots, E_{i_m})$$

conjecturar

$$P_i = \mathcal{E}_i(P_{i_1}, \ldots, P_{i_m})$$

etc.

Assim:

- Processo de verificação estruturado em *árvore*.
- Refinamento gradual tipo 'inventa & verifica'.

Sumário

Tem-se então:

$$\text{Validação} \begin{cases} \text{Teste} \\ \text{Verificação Formal} \begin{cases} \text{Estática ('a posteriori')} \\ \text{Construtiva} \end{cases} \end{cases}$$

Comentários

- Nem todas as linguagens de programação são "boas" para a verificação construtiva.
- É preciso que cada combinador sintático $\mathcal{E}$ tenha o seu combinador matemático ($\varphi_{\mathcal{E}}$).
- O porquê do célebre "horror aos `goto`'s":

$$\begin{aligned} [\![\texttt{if } x \texttt{ then } y \texttt{ else } x]\!] &= \varphi_{\texttt{if}}(x, y, z) \\ [\![\texttt{while } x \texttt{ do } y]\!] &= \varphi_{\texttt{while}}(x, y) \\ [\![\texttt{goto } x]\!] &= ?(x) \end{aligned}$$

Inconvenientes de “Inventa & Verifica”

- Inventar bem há pouco quem...
- Provar bem há pouco quem...
- Dimsensão dos problemas da vida real!

Conjectura:

- Não será possível *derivar* equacionalmente (deduzir) as implementações a partir das respectivas especificações?
- A ideia é desenvolver um *calculus* para que *programas* possam ser de facto “calculados” dedutivamente a partir de especificações.
- A redução do ónus da prova matemática obtem-se fazendo cálculo estrutural em lugar de demonstrações a partir de princípios básicos.

Relembrando o passado — *cf.* o cálculo diferencial e integral, a álgebra linear, *etc.*:

> “Quien sabe por Algebra, sabe scientificamente.” *Pedro Nunes (1502-1578) in libro de algebra, 1567, fol 270v.*

Profecia

Cálculo SETS, Cálculo de Oxford, ... — é tentador antecipar que os anos 90 venham a assitir à fase de maturação da tecnologia do ‘software’, desenvolvendo e usando ferramentas a nível industrial baseadas em processos de **cálculo de programas**.

Completar-se-á assim a evolução das técnicas de controlo de qualidade em ‘software’:

$$\text{Validação}\left\{\begin{array}{l}\text{Teste}\\ \text{Verificação}\left\{\begin{array}{l}\text{Estática (‘a posteriori’)}\\ \text{Construtiva}\end{array}\right.\\ \underline{\text{Cálculo}}\end{array}\right.$$

Queda na Realidade

- **Compatibilidade** — Os métodos formais não podem pretender substituir irrealisticamente todas as outras tecnologias da programação ($\Rightarrow$ custos, riscos, formação *etc.*)
- **Simplicidade** — a indústria dificilmente absorverá metodologias formais baseadas em teorias matemáticas muito sofisticadas;
- **Flexibilidade** — o verdadeiro desafio é proporcionar à indústria ferramentas formais que se consigam combinar com ferramentas convencionais;
- **Disponibilidade** — essas ferramentas devem ser “simples” de usar e devem estar disponíveis em múltiplas plataformas;
- **Modularidade** — o “senso comum” da vida do dia-a-dia leva-nos a resolver problemas complicados por combinação de soluções encontradas para os seus sub-problemas; o ‘software’ não deverá ser a eterna excepção a tal estratégia;
- **Reutilização** — se as soluções para sub-problemas estiverem facilmente acessíveis em repositório, então a modularidade conduzirá à reutilização de forma muito natural;
- **Regresso às origens** — na Física dos bancos de escola aprende-se uma estratégia universal para resolver problemas:
  - compreensão do problema,
  - criação de um modelo matemático,

– raciocínio no modelo,
– cálculo de uma solução

que ainda hoje é estranha à disciplina do 'software' e tem de deixar de o ser.

Analogia com Restantes Tecnologias

O "mito" da fábrica de 'software':

Nível	**Característica**	**Fábrica Convencional**	**Fábrica de 'Software'**
Produto	componente	peça bloco	modelos-ERA?
Produção	compositionalidade	árvores de produto linhas de montagem	*makefiles*?
	conhecimento acumulado	'stock'	bibliotecas ?

mas. . .

- . . . o que "é" um componente de 'software'?
- . . . como obter compositionalidade ao nível do código ?
- . . . o que é uma *fábrica de 'software'* ? [*Eureka SF 89*]

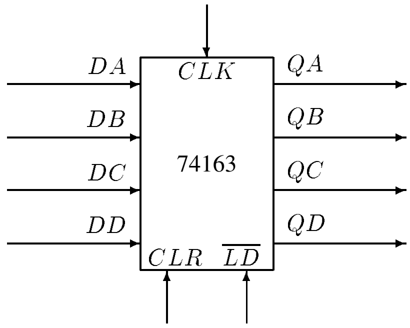
Sugestões

- Aprender lições do passado;
- Compreender o 'software';
- Formalizar os resultados.

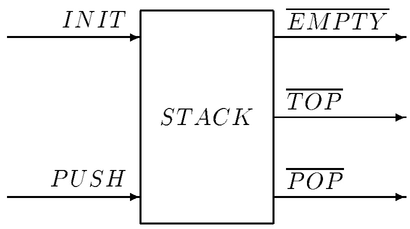
A Abordagem CAMILA/SETS

Noção de RSC (='Re-usable Software Component')

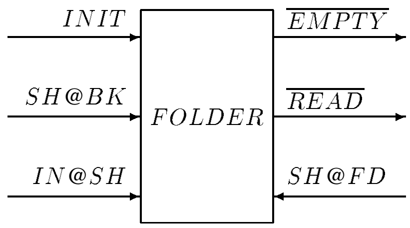
Comparar, por exemplo...



com...

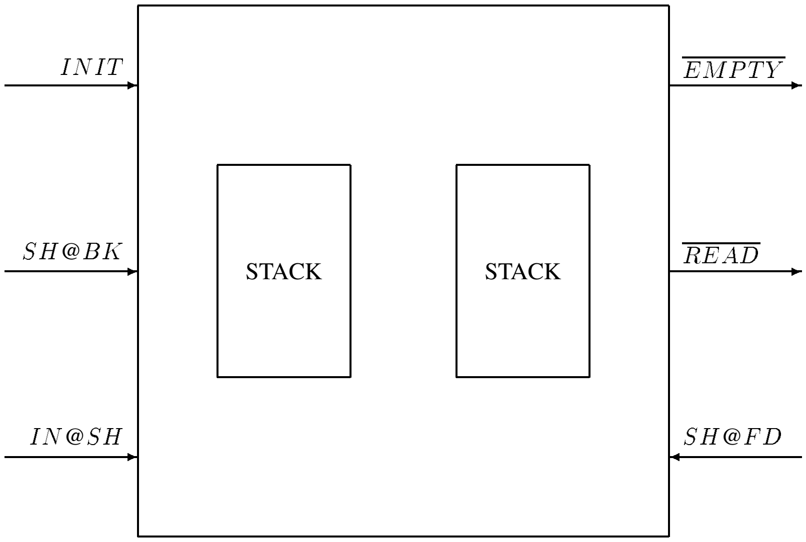


Suponhamos que o que se pretende é...

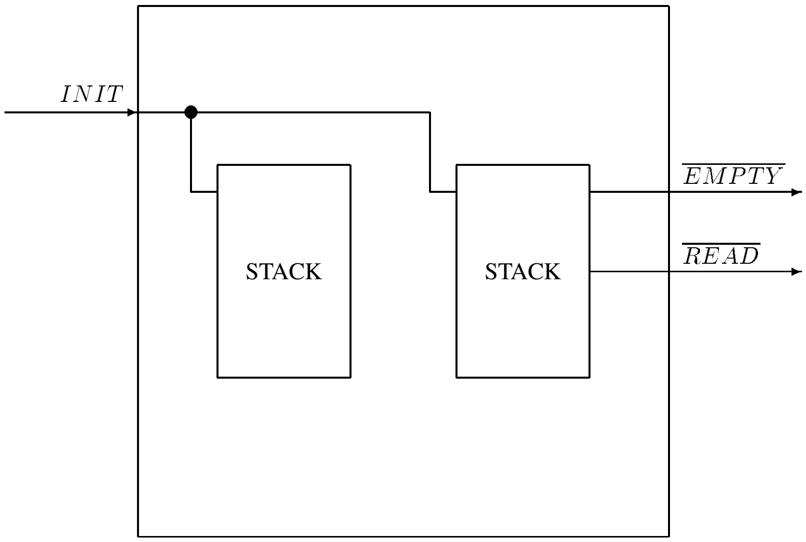


Reutilização:

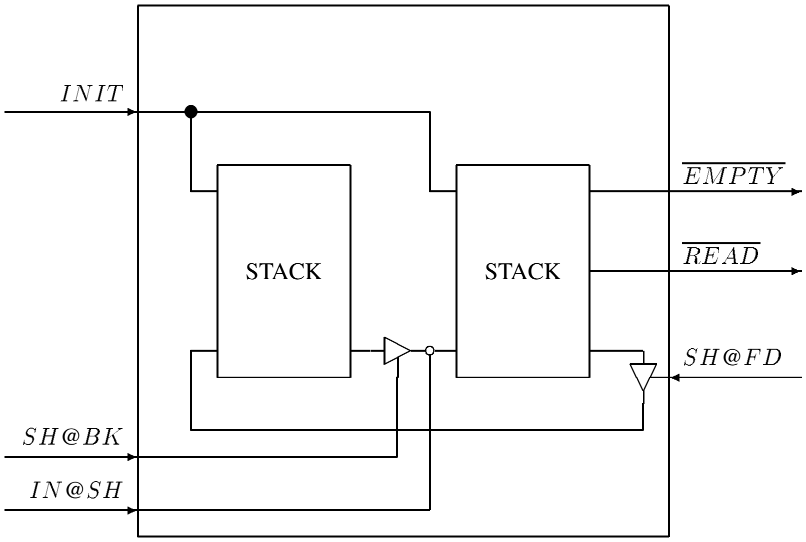
Não começar “do zero”! Faça-se “zoom in” e reutilize-se $STACK$:



Agora algumas “ligações óbvias”. . .



. . . outras mais elaboradas. . .



. . . e é tudo!

Em suma:

$$RSC = \begin{cases} \text{Componente Elementar } (\textit{e.g. } STACK) \\ ou \\ \text{Bloco = rede de } RSC\text{s } (\textit{e.g. } FOLDER) \end{cases}$$

e

$$RSC = \begin{cases} \text{interface} \quad (\text{ syntaxe }) \quad \text{— “caixa preta”+“pinos”} \\ \qquad\qquad + \\ \text{conteúdo} \quad (\text{ semântica }) \quad \text{—}\begin{cases} \text{Componente - . . . ?} \\ \text{Bloco - . . . ?} \end{cases} \end{cases}$$

Pontos nobres

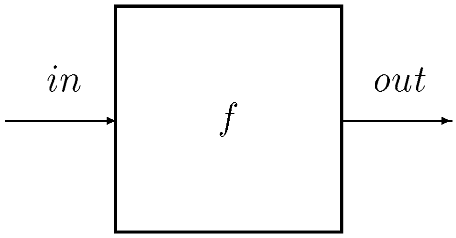
- O principal mérito da **especificação formal** de ‘software’ é o de responder às “...” acima de forma que torna possível **raciocinar** matematicamente sobre RSCs.
- Programar não é uma arte — é uma técnica!

$$\text{Programar} = \left\{ \begin{array}{lcl} \text{Especificar} & = & \text{gesto criativo (SETS)} \\ & + & \\ \text{Prototipar} & = & \text{“ver para crer...” (CAMILA)} \\ & + & \\ \text{Refinar} & = & \text{processo dedutivo (SETS)} \end{array} \right.$$

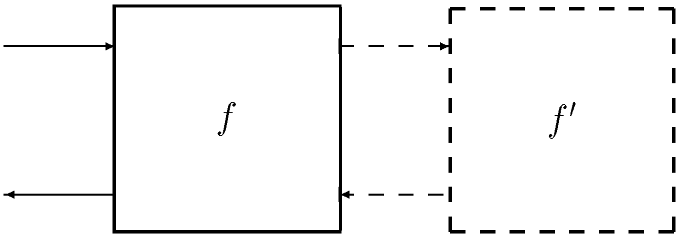
- Prototipagem “não oca” — pôr toda a semântica a correr e não apenas um simulacro de IPM.

Ciclo de Vida

Perspectiva da aplicação do ponto de vista do *cliente*:



A aplicação do ponto de vista do *servidor* — raríssimas vezes uma funcionalidade é autosuficiente:

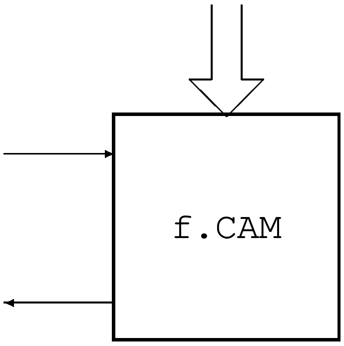


Requisitos, especificação formal e protótipo:

Pretende-se uma aplicação para a automatização do sistema de gestão da produção da fábrica ....
bla. . . bla. . . bla. ....

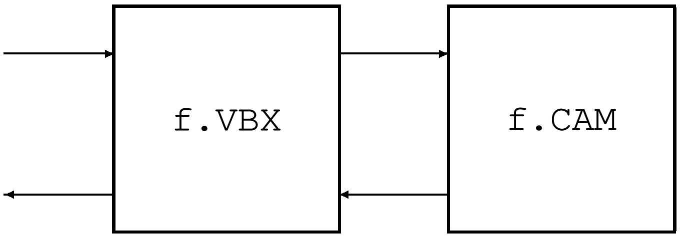
f.DOC = ...
....
....
....

$$\text{f.SETS} = \begin{cases} s = x \times l \\ x = b \times (x+1)^a \\ l = 1 + a \times l \end{cases}$$

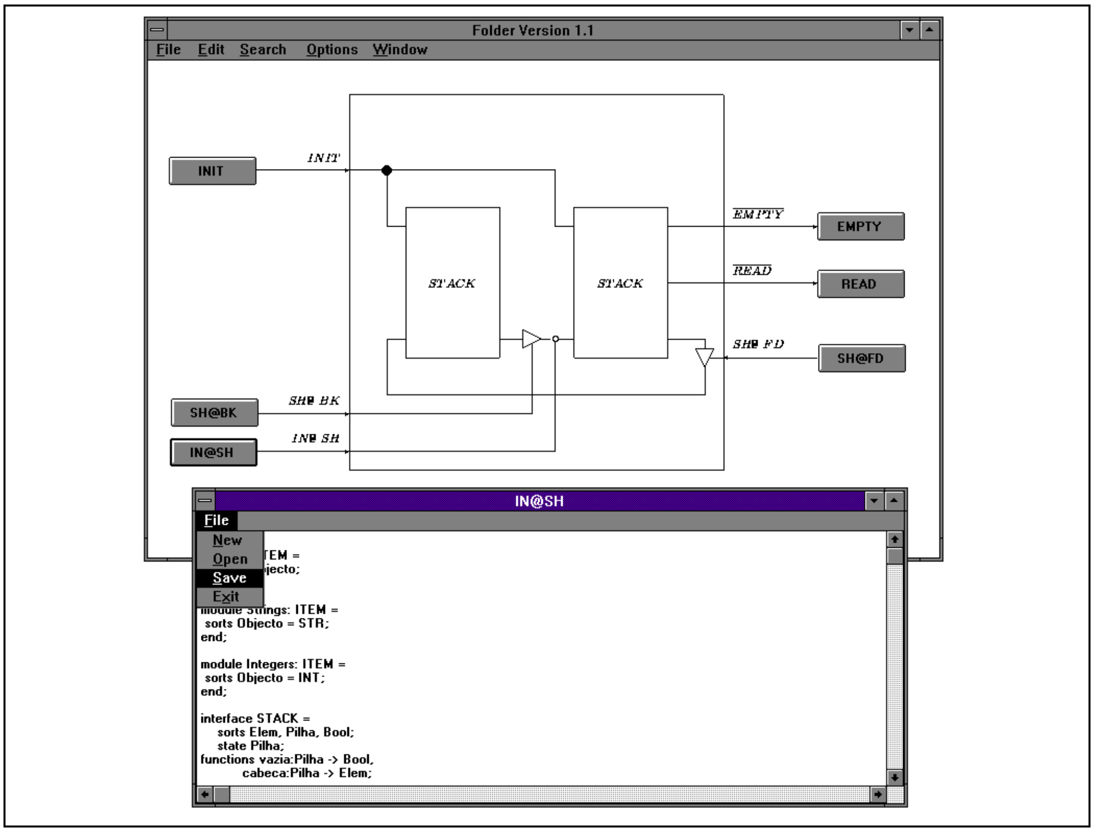


Sofisticação da Interface com Utilizador. . .

. . . para “enganar o cliente”:

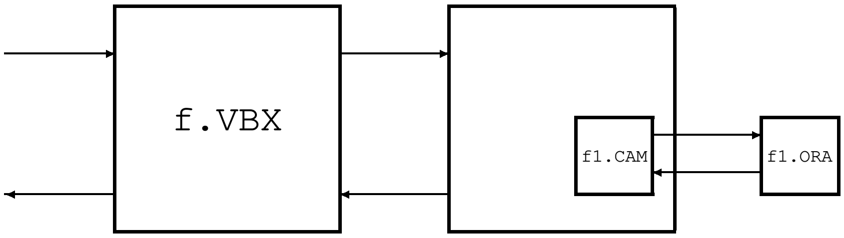


o que se pode revelar muito útil:

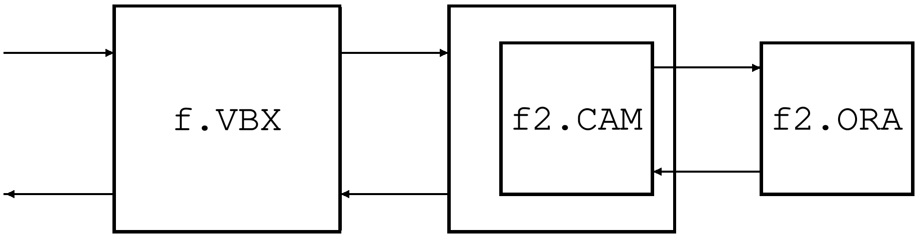


Implementação

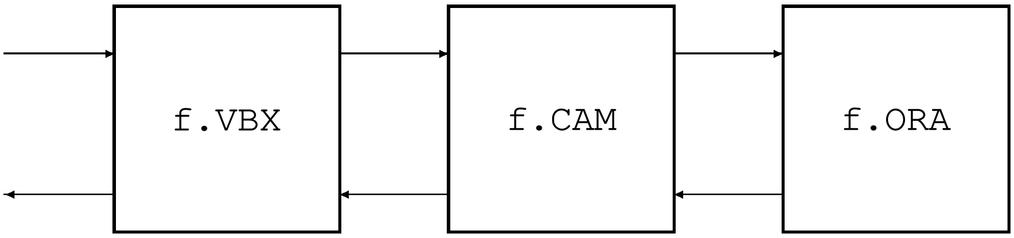
Quando agradar, comece-se a calcular a implementação, . . .



. . . gradualmente. . .



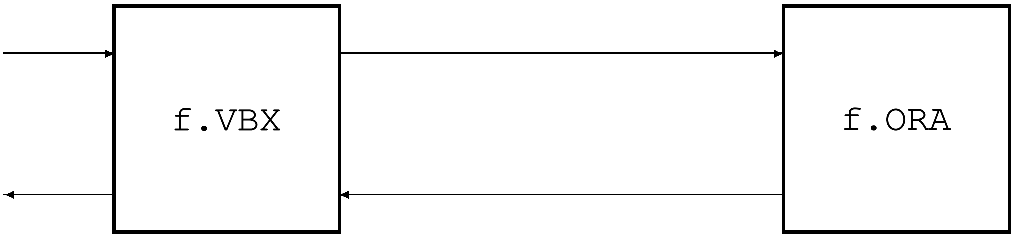
até. . .



Finalmente, o. . .

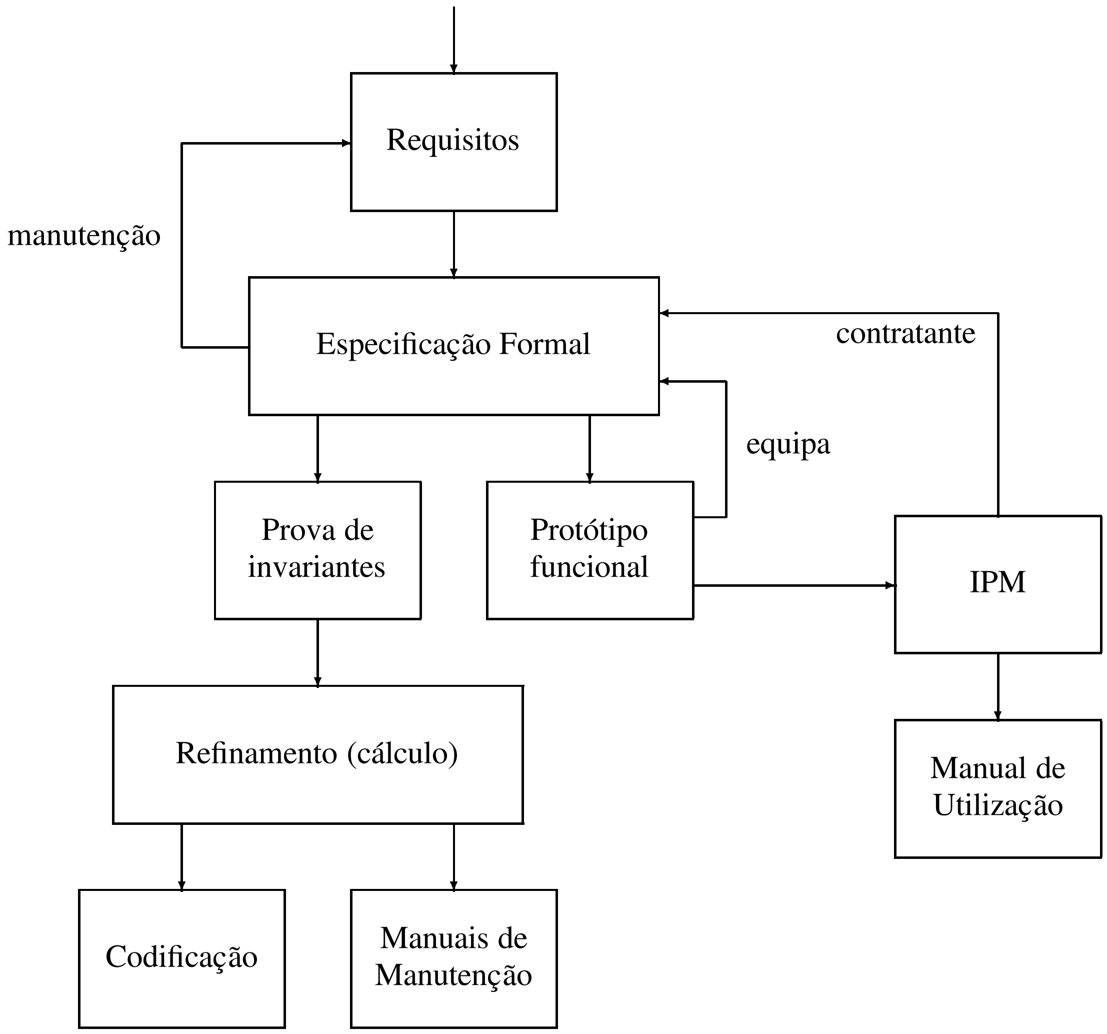
Acabamento

. . . ou o fim do protótipo:



incluindo toda a codificação.

Sumário do Ciclo de Vida



Comentários:

(a) Ciclos de desenvolvimento “curtos” concentram-se à volta da *especificação formal* e longe do código.

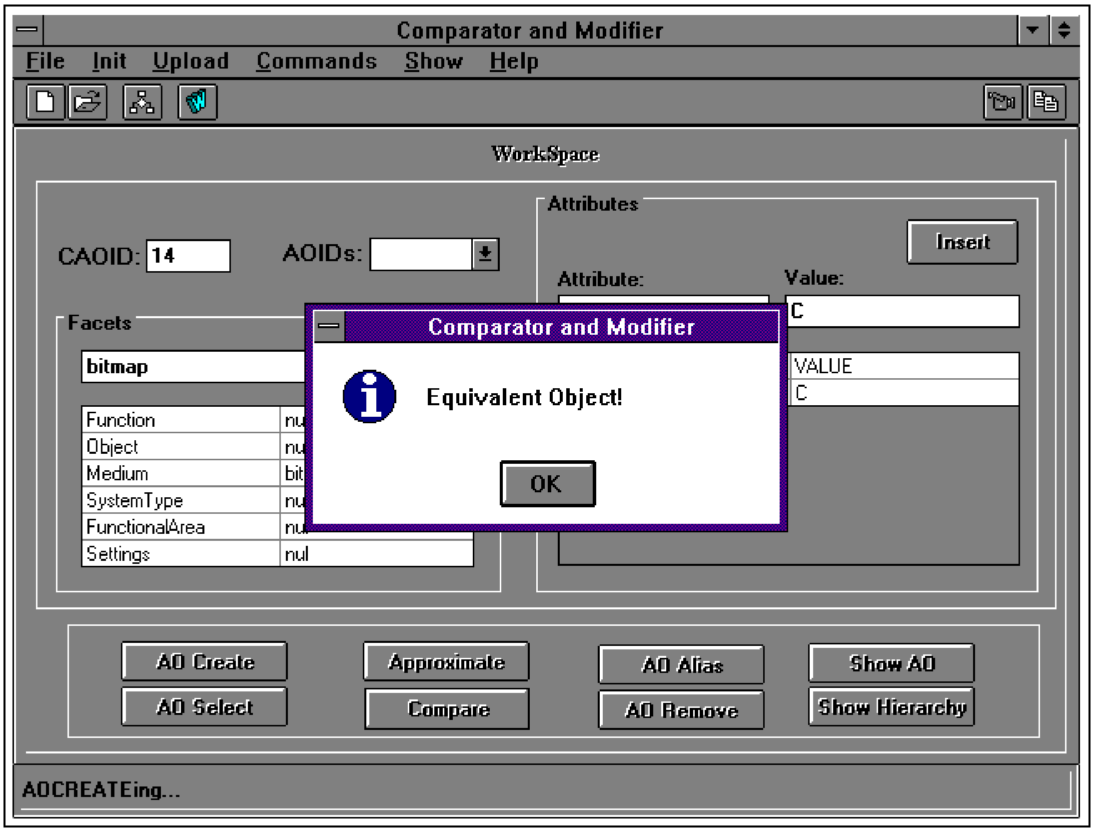
(b) Prototipagem rápida integral esconde-se sob a camada de apresentação “definitiva”, *cf.* écran da transparência 18.

(c) Cálculo científico e gradual da implementação (a *qualidade* desejada).

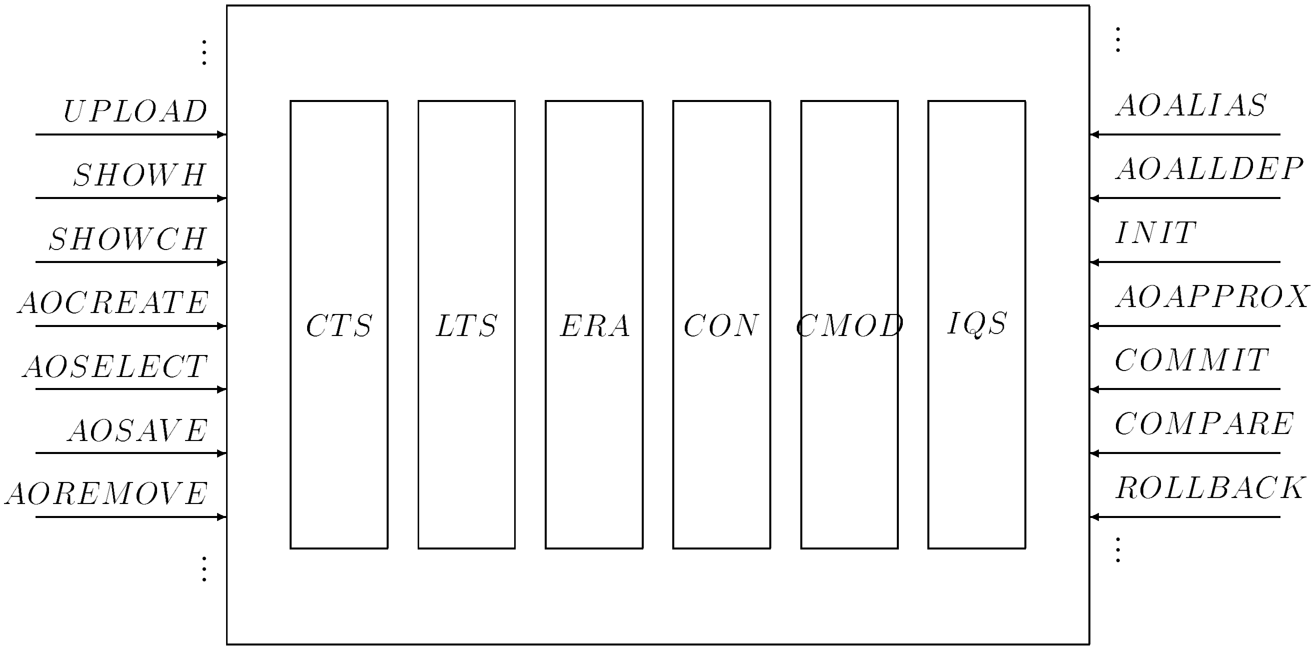
(d) Produção "inerente" e sistemática de documentação.

De Volta ao Caso na Indústria

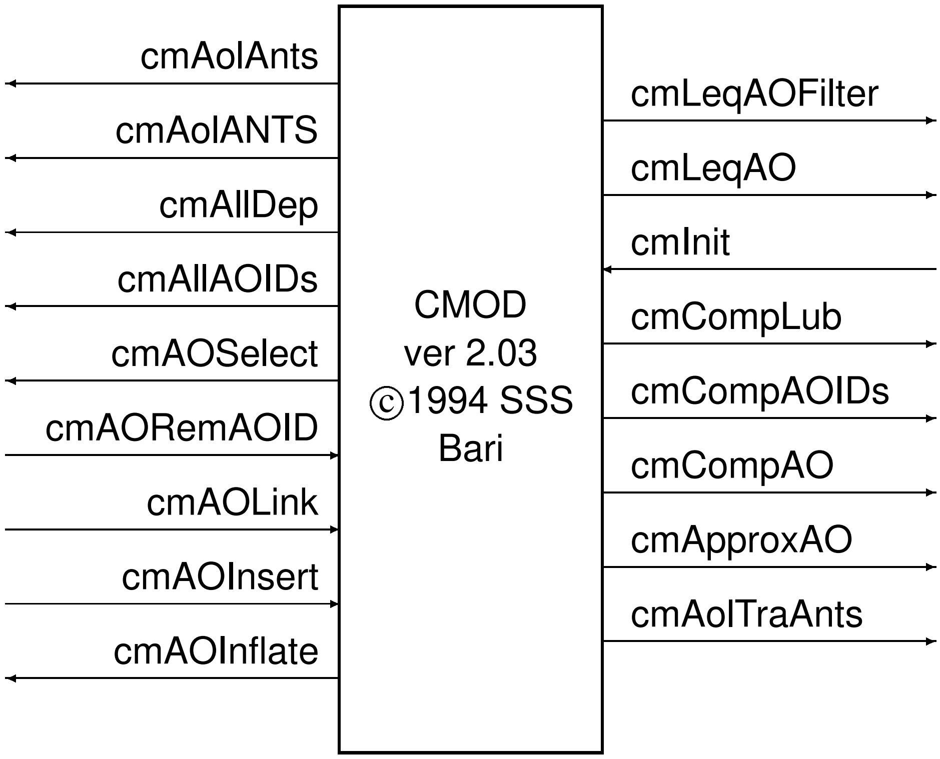
Especificando formalmente em SETS e usando a prototipagem CAMILA foi possível apresentar em Roma (Jan. 1994) "software a correr", quando apenas se esperavam especificações funcionais, *e.g.*:



cuja arquitectura interna é, 'grosso modo', a da seguinte "placa de 'software"':



Exemplo de um "integrado de 'software'" desenvolvido em Braga para a firma *Sss*:



NB: Os módulos

- *Era* (Easy Repository Access),
- *Lts* (Lexicon Thesaurus Subsystem) e
- *Cts* (Concept Thesaurus Subsystem)

— na altura ainda a serem desenvolvidos em paralelo pela empresa *Systena* — foram temporariamente *reverse engineered* para CAMILA/SETS a partir das respectivas f. specs.

Contudo...

A utilização de métodos de especificação formal e do CAMILA não foi oficial.

Alguns Números do SOUR

Em Braga desenvolveram-se, até ao momento,

- linhas de código CAMILA, incluindo especificações reversas dos módulos italianos: .... $\simeq$ 4 200
- linhas de código C/Visual Basic: .... $\simeq$ 31 400

CAMILA na Realidade

Simplicidade

A linguagem de especificação da abordagem CAMILA tem um sabor funcional e baseia-se em construtores da teoria ingénua dos conjuntos estudada ao nível do ensino liceal — *cf.* Quadro da transparência 24.

$SETS$	PASCAl	C	**Descrição**

$A \times B$	`record`
	`   P: A;`
	`   S: B`
	`end;`
		`struct X {`
		`   A first;`
		`   B second;`
		`};`
			'Records'

$A + B$	`record`
	`   case`
	`    tag:  integer`
	`        of x =`
	`     1:  (P:A);`
	`     2:  (S:B)`
	`end;`
		`union X {`
		`   int tag; /* 1,2 */`
		`   A ifA;`
		`   B ifB;`
		`};`
			'Records' variantes

B^A	`array[A] of B`	`B[A]`	'Arrays'
$A + 1$	`Â`	`A *...`	'Pointers'

Flexibilidade e Disponibilidade

- A **ferramenta de prototipagem rápida** disponível no sistema tem cidadania plena ao nível da programação em C/C++ (C pode chamar CAMILA e CAMILA pode fazer chamadas a C externo) e está disponível tanto em MS/DOS como em UNIX.
- Qualquer protótipo CAMILA pode prestar serviços em X-Windows ou funcionar como uma DLL Windows 3.1.
- A camada de interação pessoa-máquina (IPM) é sempre programada independentemente do protótipo da aplicação.
- Está em curso o desenvolvimento de um módulo para especificação da camada IPM e geração automática de código Visual C++/Visual Basic ou TCL/TK.

Modularidade

- Uma versão da linguagem de especificação CAMILA é modular, correspondendo cada protótipo à tradução de uma expressão modular envolvendo componentes disponíveis em repositório ou especificados localmente, que dialogam entre si através de um mecanismo de interfaces formais.
- A agregação de componentes pode ser expressa numa notação gráfica sugestivamente parecida com a que se usa em projecto de 'hardware' mas com uma semântica matemática bem definida.

Reutilização

- Reutilização de *especificações formais* ($\Rightarrow$ por transitividade, das respectivas *implementações*).

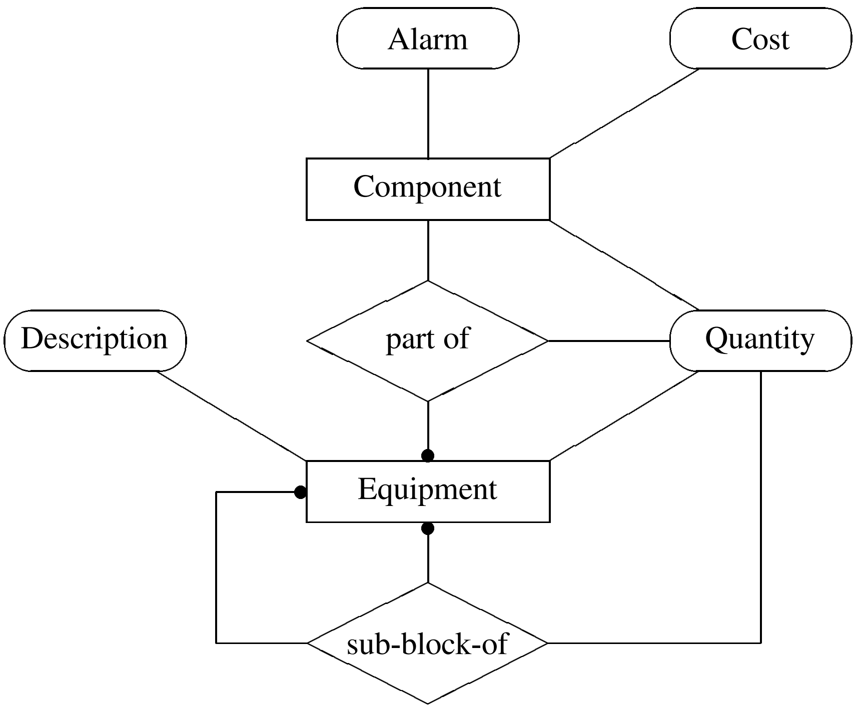
- O repositório de componentes CAMILA está a basear-se num **critério formal** de **classificação** baseado no cálculo acima descrito.

Complementaridade

Ou a necessária compatibilidade com métodos tradicionais, relembrar 'mea culpa'. Apenas dois exemplos:

Exemplo 1: Diagramas-ERA são Modelos Formais de Dados

Exemplo: o seguinte diagrama para gestão de *stocks de peças e ŕvores de produtos*



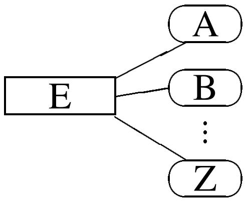
— retirado de [*Oliveira & Cruz 93*] — converte-se no seguinte modelo formal de dados:

$$
\begin{array}{lll}
(& (\#Equipment \rightarrow & Description \times Quantity \times \\
 & & ((\#Equipment + \#Component) \rightarrow Quantity) \\
 & (\#Component \rightarrow & Alarm \times Cost \times Quantity) \\
)_\varphi & &
\end{array}
\quad) \times
$$

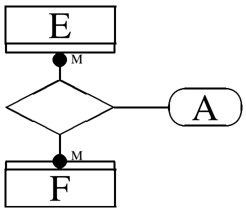
sujeito ao *invariante* (restrição de integridade):

$$
\begin{array}{lll}
\varphi(e_1, e_2) \stackrel{\mathrm{def}}{=} & let & X = \bigcup_{k \in dom(e_1)} dom(\pi_3(e_1(k))) \\
 & & E = \{x \in \#Equipment \mid \langle 1, x \rangle \in X\} \\
 & & C = \{x \in \#Component \mid \langle 2, x \rangle \in X\} \\
 & in & dom(e_2) = C \wedge E \subseteq dom(e_1)
\end{array}
$$

Processo de conversão — ver regras e cálculo de estrutura de dados em [*Oliveira & Cruz 93*], *e.g.*:

- 

$$= \#E \rightarrow A \times B \times \cdots \times Z \qquad (2)$$

- 

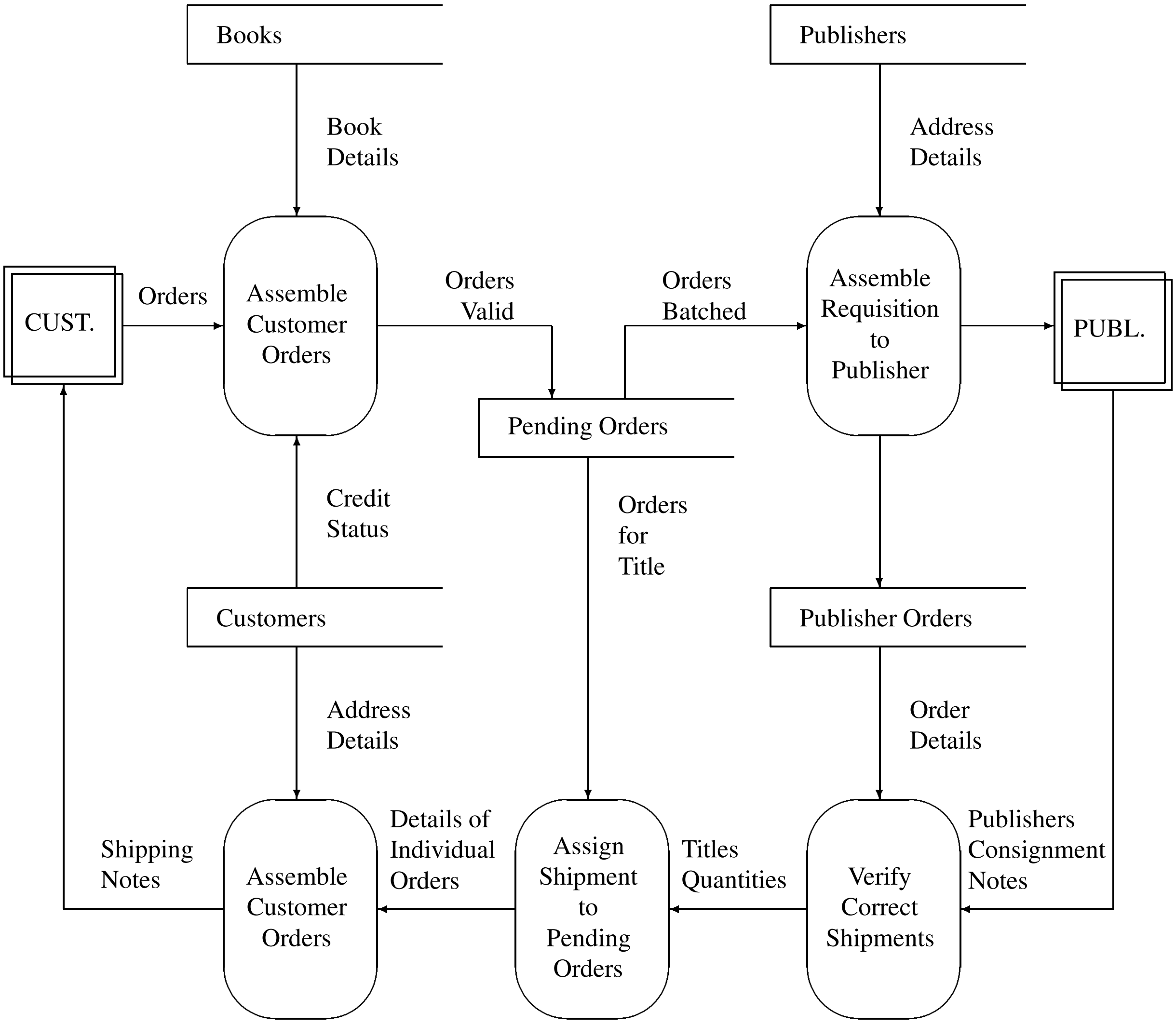
$$
= \quad \begin{array}{l} ((\#E \times \#F \rightarrow A) \times \\ (\#E \rightarrow \cdots) \times \\ (\#F \rightarrow \cdots))_\varphi \end{array} \qquad (3)
$$

  Invariante associado:

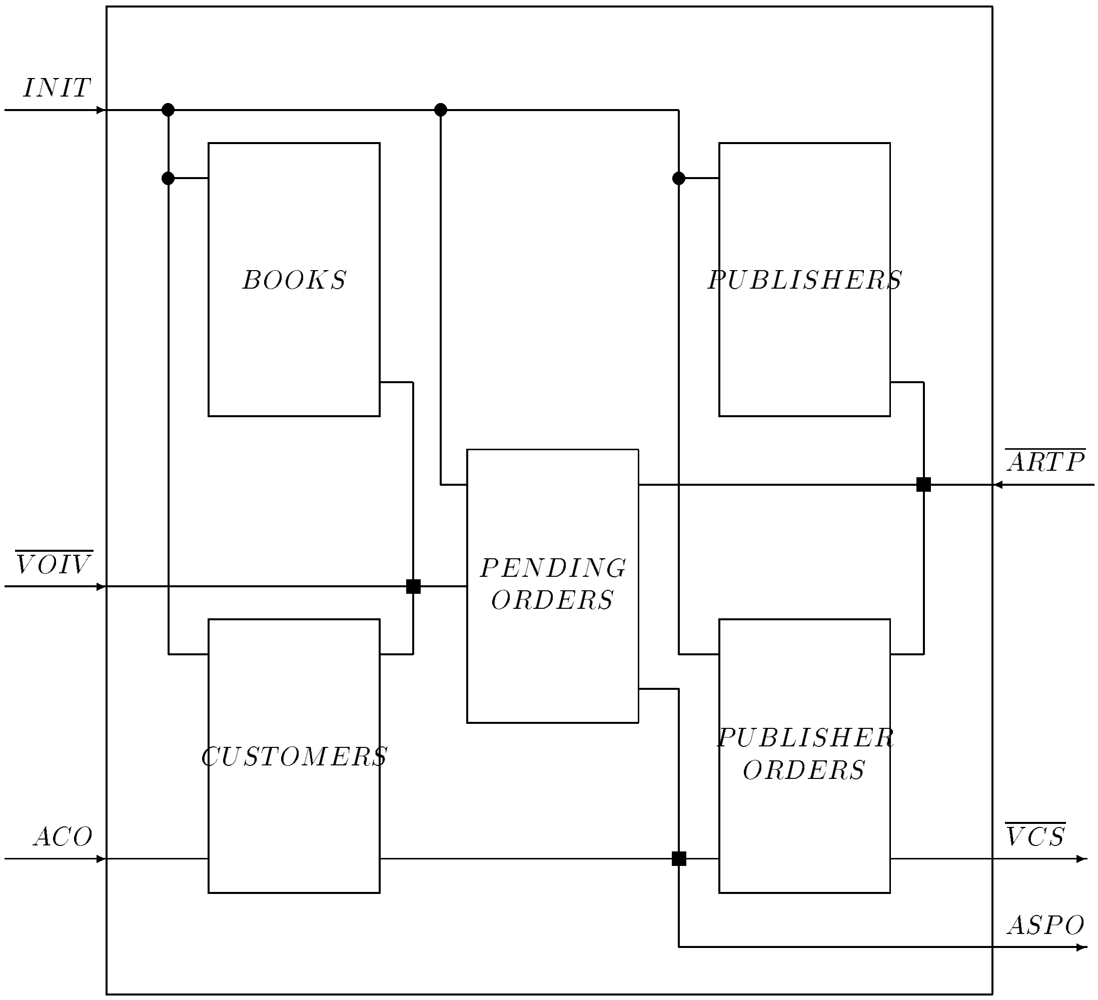
$$
\begin{array}{ll}
\varphi(\rho, \sigma, \sigma') \stackrel{\mathrm{def}}{=} & \pi_1[dom(\rho)] \subseteq dom(\sigma) \wedge \\
 & \pi_2[dom(\rho)] = dom(\sigma')
\end{array}
$$

Exemplo 2: ‘DFD’s são “Placas” de ‘Software’

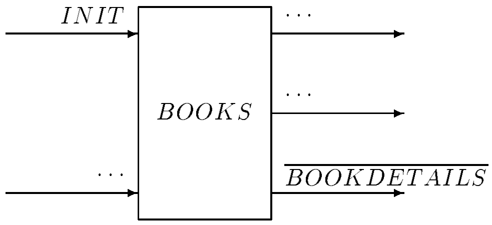
Um exemplo conhecido [*Gane & Sarson 79*]:

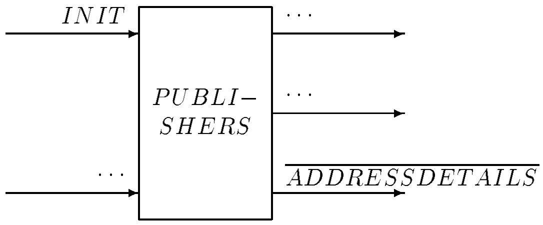


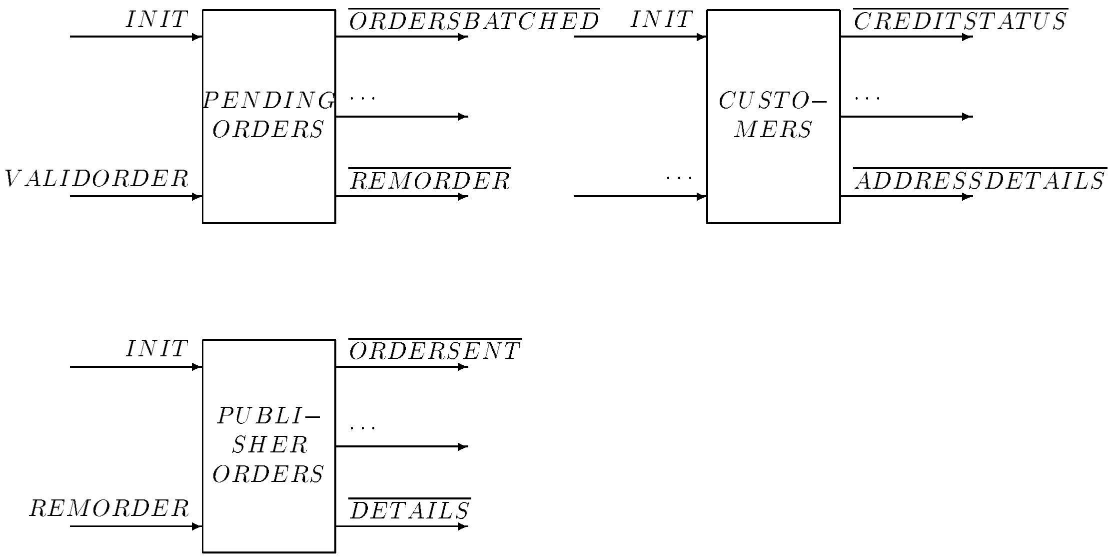
e sua conversão para a notação gráfica CAMILA:



envolvendo os “integrados”:







Apenas um breve “zoom” sobre a semântica de $VOIV$ (‘Verify Order is Valid’) em sintaxe concreta CAMILA, de claro “sabor funcional”:

```
;----------------------------------------------------------
; Verify Order is Valid

VOIV: Order -> SYM
VOIV(o) = let (c=N(o),
               S= bds.BOOKDETAILS(B(o))
               )
          in if card(S) == 1
             then let (i=the(S),
                         x=bds.GET(i),
                         a=C(x),
                         p=P(x))
                  in if (cds.CREDITSTATUS(c) >= a)
                     then pds.VALIDORDER(VOrder(c,i,p))
                     else error("No credit, sorry!")
              else error("Unknown book or insufficient details");
;----------------------------------------------------------
```

Regresso às origens

A tecnologia da programação em CAMILA tem como ponto nobre um **cálculo de programas** — SETS, *cf.* [*Oliveira 1990*], [*Oliveira 1992*], *etc.* — que permite partir de uma especificação e, por transformações suas, obter o código executável final, com garantia automática de correcção.

Notas Finais

- 10 anos (1984–94) de investigação e ensino, na Universidade do Minho, de *métodos formais de programação*.
- **Métodos tradicionais** são deficientes — *e.g.* onde está, descrito no diagrama-ERA que atrás se referiu, a indicação de que *"nenhum equipamento pode recursivamente ser sub-bloco de si próprio"*, essencial para que a operação de *explosão em peças* termine?
- O software CAMILA (ver transparente 34) está ainda em desenvolvimento.
- Tem sido objecto de acções regulares de formação em Bristol (UK), e ainda em Pernambuco (Brasil) e Zielona Gora (Polónia).

Bibliografia

[Almeida & Barbosa 91] Almeida J., Barbosa L.S. *CAMILA by Example.* Lecture Notes, Technical Report 91-11-1, DI/INESC, November 1991.

[Barroca 89] Barroca L. *Formal Models for Systems Prototyping.* Ph.D. thesis, University of Southampton, 1989.

[Eureka SF 89] Eureka Software Factory *ESF Technical Reference Guide.* TDG Version 1.1, 1989.

[Gane & Sarson 79] Gane C., Sarson T. *Structured System Analysis.* Prentice-Hall, 1979.

[Jones 80] Jones C.B. *Software Development — A Rigorous Approach.* Prentice-Hall International, Series in Computer Science (C.A.R.Hoare Ed.), 1980.

[Oliveira 90] Oliveira J.N. *A Reification Calculus for Model-Oriented Software Specification.* Formal Aspects of Computing, Vol.2, 1-23, 1990, Springer-Verlag.

[Oliveira 92] Oliveira J.N. *Software Reification using the SETS Calculus.* Invited communication. Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK, 8–10 January 1992, 140-171, Springer-Verlag.

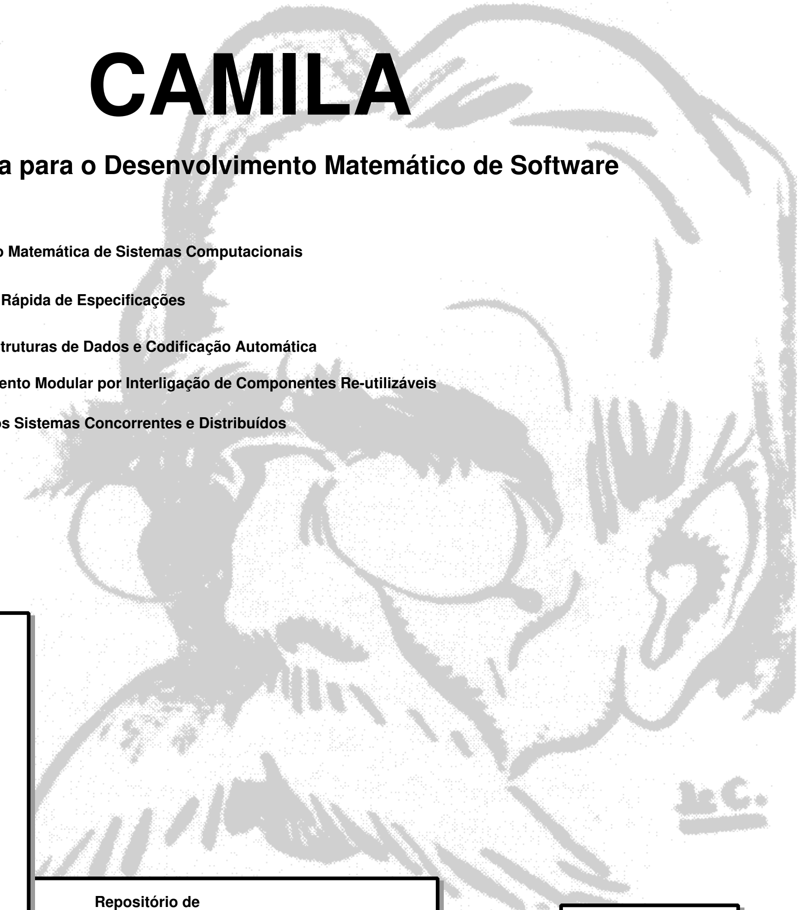
[Oliveira & Cruz 93] Oliveira J.N., Cruz A.M. *Formal Calculi Applied to Software Component Knowledge Elicitation*. Technical Report C19-WP2d, Project *C.1.9. Sviluppo di Metodologie, Sistemi e Servizi Innovativi in Rete*, December 1993.

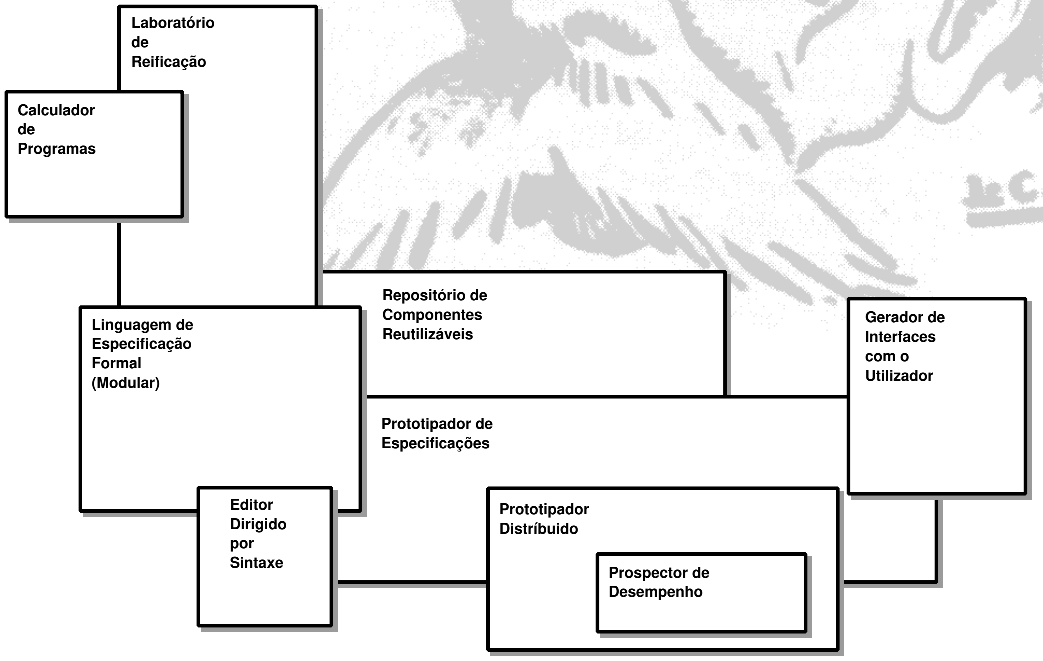
[Vicente 94] Vicente B. *Qualidade em Sistemas de Informação.* Tutorial I, Seminário TSI'94, Idite-Minho, Braga, 23–25 Maio 1994.

CAMILA

Plataforma para o Desenvolvimento Matemático de Software

- Especificação Matemática de Sistemas Computacionais
- Prototipagem Rápida de Especificações
- Cálculo de Estruturas de Dados e Codificação Automática
- Desenvolvimento Modular por Interligação de Componentes Re-utilizáveis
- Orientação aos Sistemas Concorrentes e Distribuídos





CAMILA

Linha de Acção: ***Fundamentos da Programação***

Projecto: ***DI - INESC***

Coordenação: ***Prof. José Nuno Oliveira***

Apoio: ***JNICT - PMCT***

Estou agora a ver se ajunto umas migalhas para comprar um cardenho entre dois carvalhos aí para o coração do Minho, e fechar-me lá, resguardar-me dos lobos e dos homens e das mulheres, com duas canastras de livros.

Camilo Castelo-Branco

(O acrónimo CAMILA celebra o (quase esquecido) Centenário da sua morte, em 1990)