Construção e Teste de Modelos Executáveis Integrando VDM++ e UML

João Pascoal Faria
jpf@fe.up.pt www.fe.up.pt/~jpf

UCE30 Métodos Formais em Engenharia de Software,
Universidade do Minho, Braga, 24 e 31 de Janeiro de 2008

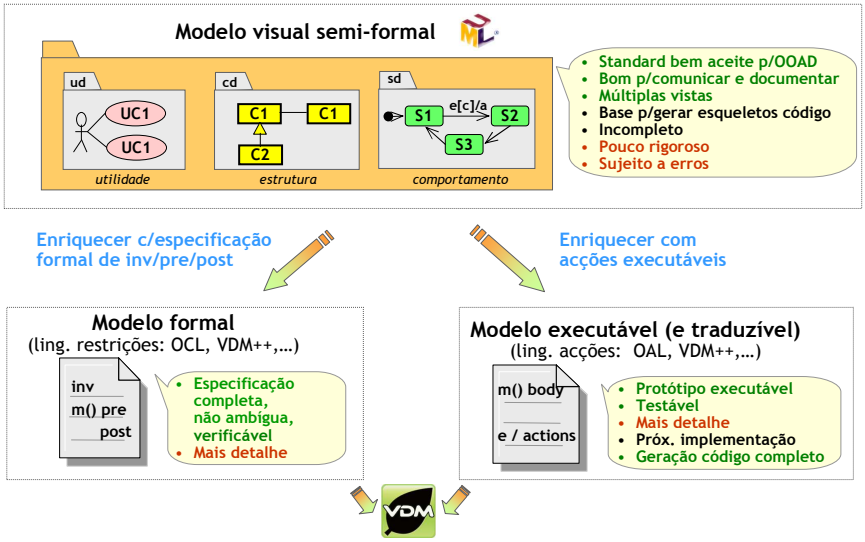
Plano

- Pré-condições:
  - Conhecimentos básicos de VDM-SL, UML e OOP
- Escalonamento:
  - 24/1/08, 13-16h, Teoria, Parte I - modelação de estrutura, espec. inv/pre/post
  - 31/1/08, 13-16h, Teoria, Parte II - modelos executáveis, testes, asp. dinâmicos
  - 31/1/08, 16-19h, Prática - VDM Tools, Rational Rose, projecto
- Pós-condições:
  - Saber construir um modelo formal executável em VDM++
  - Saber validar o modelo através de testes automatizados em VDM++
  - Saber integrar diagramas UML com especificações em VDM++
  - Conhecer as capacidades de round-trip engineering c/ Rational Rose e VDM Tools
  - Conhecer a aplicabilidade, vantagens e limitações das técnicas exploradas

Índice

Motivação

De UML a VDM++



Modelos visuais (semi-formais) em UML

- Diagramas UML são muito usados nas fases análise de requisitos e desenho do sistema, para comunicar e raciocinar sobre requisitos e decisões de desenho
- Geração de algum código (Java, C#, SQL, XSD, etc.) a partir de UML (esqueletos de código e não código completamente funcional)
- Problemas: informais, incompletos, muito sujeitos a erros
- Podem ser enriquecidos em duas direcções ortogonais (mas combináveis):
  - No sentido de chegar a um modelo formal
  - No sentido de chegar a um modelo executável (e traduzível)

Modelos formais

- É possível enriquecer os modelos visuais (semi-formais) com especificações formais de
  - restrições de estado (por invariantes)
  - semântica de operações (por pré-condições e pós-condições)
- A própria norma UML define uma linguagem para este efeito: OCL (Object Constraint Language)
- Obtém-se um modelo formal que funciona como especificação rigorosa (sem ambiguidades, inconsistências ou omissões), detalhada e verificável do sistema
  - A especificação formal remove ambiguidades da especificação informal (embora à custa de maior detalhe)
  - A especificação formal é verificável por máquinas: por exemplo, existem ferramentas que geram asserções em Java a partir de especificações de invariantes, pré-condições e pós-condições em OCL

Modelos executáveis

- É possível enriquecer os modelos visuais com especificações do corpo algorítmico de operações (bem como de acções e actividades) em linguagens de acções de alto nível
- A própria norma UML define a sintaxe abstracta (capacidades) de uma linguagem de acções de alto nível (UML Action Semantics)
  - A linguagem concreta não é fixada pela norma, mas existem várias linguagens concretas compatíveis com a norma
- Obtém-se um modelo executável, que serve como protótipo executável do sistema, permitindo testar e validar precocemente validar requisitos e opções de design

Modelos traduzíveis

- Os modelos executáveis são também facilmente traduzíveis (por um compilador de modelos) para uma linguagem de implementação alvo
- Estamos a falar de geração automática de código completamente funcional e não só esqueletos de classes
- Particularidades das linguagens, tecnologias e plataformas-alvo são embebidas nos geradores (conceito MDA – Model Driven Architecture)
- Vantagens: Aumento de produtividade, foco no domínio do problema e não nas tecnologias de implementação
- Desvantagens: Código gerado pode ser pouco eficiente, continuando a ser necessário escrever código na linguagem-alvo!

Modelos visuais, formais, executáveis e traduzíveis: que solução integrada?

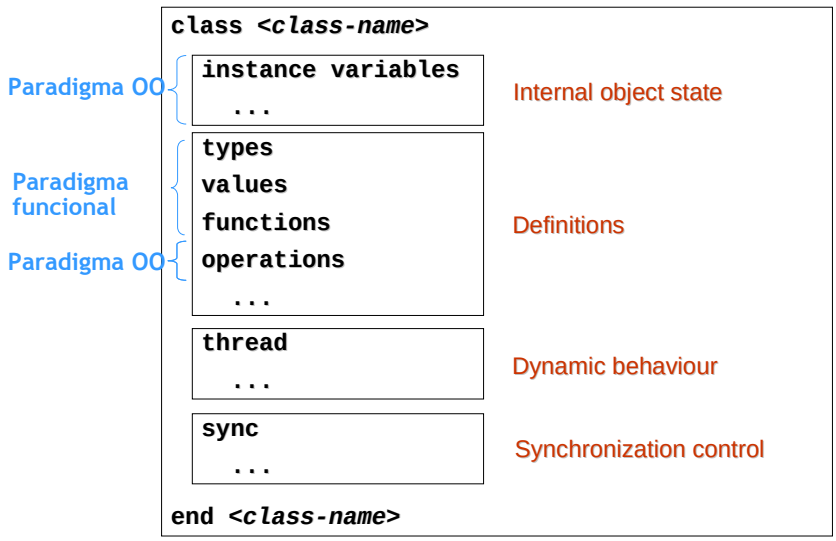
- Em torno da norma UML têm surgido linguagens que permitem criar modelos formais - caso de OCL - e modelos executáveis e traduzíveis – como xUML - mas ainda não de forma perfeitamente integrada
- Em contrapartida, VDM++ é uma linguagem de especificação formal OO que permite criar modelos formais, executáveis e traduzíveis, sendo suportada por ferramentas (VDMTools) que permitem executar e testar os modelos, sincronizar com Rational Rose (diagramas UML) e gerar código Java e C++
- OCL e xUML terão mais importância no futuro, no entanto VDM++ é presentemente uma solução mais madura, integrada e suportada por ferramentas

Estrutura geral duma especificação em VDM++

Características gerais de VDM++

- Baseada no *standard* VDM-SL
- Linguagem de especificação formal baseada em modelos (i.e., com representação explícita de estado) orientada por objectos
- Combina dois paradigmas
  - Paradigma funcional: tipos, funções e valores (instâncias de tipos)
  - Paradigma OO: classes, variáveis de instância, operações e objectos (instâncias de classes)
- Suportada por ferramentas VDMTools que permitem:
  - Executar uma especificação escrita em VDM++
  - Testar a especificação e analisar a cobertura dos testes
  - Sincronizar com diagramas de classes UML na ferramenta Rational Rose
  - Gerar código Java e C++
- Disponível em duas notações: ASCII ou símbolos matemáticos

Estrutura duma especificação em VDM++



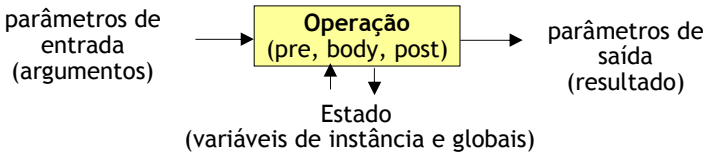
Classes

- Uma especificação em VDM++ é organizada em classes
- Classes são "tipos referência" (*reference types*)
  - Tal como na generalidade das linguagens OO
  - Instâncias são objectos mutáveis acessíveis por uma referência
  - Variável do tipo *C*, em que *C* é uma classe, contém uma referência para o objecto com os dados, e não os próprios dados
  - Comparação e atribuição operam com referências
- Usar para modelar o estado do sistema
  - Estado é representado pelo conjunto de objectos existentes e pelos valores das suas variáveis de instância
  - Classes representam tipos de entidades físicas (pessoa, livro, sala, …), papéis (professor, aluno, …), acontecimentos (aula, …), documentos (factura, contracto, …), etc.

Variáveis de instância

- Correspondem a atributos em UML e campos em Java e C#
- Podem ser *private* (por omissão), *public* ou *protected*
- Podem ser *static* (estáticas)
- Declaradas na secção "instance variables" com a sintaxe:
  [private | public | protected] [static] *nome* : *tipo* [:= *valor_inicial*];
- Podem-se definir invariantes (inv) que restringem os valores válidos das variáveis de instância
  - A tratar mais tarde

Operações



- Correspondem a operações em UML e métodos em Java ou C#
- Podem ser *private* (por omissão), *public* ou *protected*
- Podem ser *static*
- Podem consultar ou modificar o estado de obj's (dado por var's de instância) ou o estado global do sistema (dado por var's estáticas)
- Podem ter pré-condição, corpo (definição explícita, imperativa) e pós-condição (definição implícita)

Operações - definição

- Estilo 1:

```
op(a: A, b: B, …, z: Z) r: R ==
  bodystmt
ext rd instvarx, instvary, …
    wr instvarz, instvarw, …
pre preexpr(a, b, …, instvar1, instvar2, …)
post postexpr(a, b, …, r, instvar1, instvar2, …,
                  instvar1~, instvar2~, …) ;
```

argumento; tipo; resultado; tipo; omitir quando não retorna nada

- Estilo 2:

```
op: A * B * … ==> R
op(a,b,…) ==
  bodystmt
pre preexpr(a, b,…, instvar1, instvar2, …)
post postexpr(a, b,…, RESULT, instvar1, instvar2, …,
                   instvar1~, instvar2~, …) ;
```

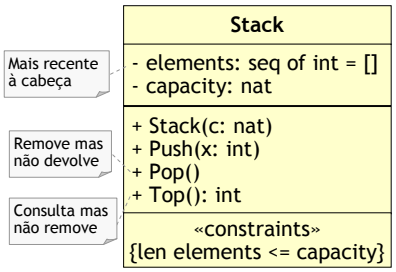
quando não há argumentos ou resultados, escrever ()

nome pré-definido para o valor retornado

Operações - partes da definição

- Pré-condição (pre) - restrição nos valores dos argumentos e variáveis de instância, a verificar na chamada
  - Pode ser omitida (mesmo que true)
- Corpo algorítmico (bodysmt) - instrução ou bloco de instruções entre ( )
  - Permite exprimir algoritmo e executar a operação (definição explícita)
  - Paradigma imperativo: c/atribuições, declaração variáveis, etc.
  - Operação abstracta: "is subclass responsibility"
  - Operação por definir: "is not yet specified", ou omitir "==bodystmt" no estilo 1
- Pós-condição (post) - restrição nos valores dos argumentos, resultado, valores iniciais ("~") e finais das var.s de instância, a verificar no retorno
  - Permite verificar o resultado/efeito da operação (definição implícita)
  - Pode ser omitida (mesmo que true)
- Cláusula "ext" (externals) - lista as variáveis de instâncias que podem ser lidas (rd) e actualizadas (wr) no corpo da operação
  - Obrigatório indicar no estilo 1, quando se indica a pós-condição

Exemplo de classe



DEMO

```
class Stack
  instance variables
    private elements : seq of int := [];
    private capacity:  nat;
    inv len elements <= capacity;
  operations
    public Stack(c : nat) res: Stack ==
      (elements := []; capacity := c)
      ext wr elements, capacity
      post elements  = []  and capacity = c ;
    public Push(x: int) ==
       elements := [x] ^ elements
      ext wr elements
      pre len elements < capacity
      post elements  = [x]  ^  elements~ ;
    public Pop() ==
      elements := tl elements
      ext wr elements
      pre elements <> []
      post elements =  tl elements~ ;
    public Top() res: int ==
      return hd elements
      ext rd elements
      pre elements <> []
      post res = hd elements;
end Stack
```

concatenar
valor antigo
tail
head

Tipos

- Tipos são “tipos valor” (*value types*)
  - Instâncias são valores puros imutáveis
  - Comparação e atribuição operam com os próprios valores
  - Variável do tipo *T* (nome de um tipo) tem os próprios dados
- Subdividem-se em:
  - Tipos básicos - bool, nat, real, char, ...
  - Tipos construídos (colecções, etc.) – set of T, seq of T, map T1 to T2, ...
- Novos tipos podem ser definidos dentro de classes na secção “types”
- Definição pode incluir invariante que restringe instâncias válidas
  - A estudar mais tarde
- Usar para modelar tipos de valores de atributos (tipos de dados)

Tipos básicos

Símbolo	Descrição	Exemplos de valores
bool	Booleano	true, false
nat1	Número natural não nulo	1, 2, 3, …
nat	Número natural	0, 1, 2, …
int	Número inteiro	…, -2, -1, 0, 1, …
rat **real**	Número racional Número real (mesmo que "rat", pois só números racionais podem ser representados no computador)	-12.78, 0, 3, 16.23
char	Carácter	'a', 'b', '1', '2', '+', '-', …
token	Encapsula um valor (argumento de mk_token) de qualquer tipo (útil quando se sabe pouco sobre o tipo)	mk_token(1)
<identificador>	*Quotes* (nomes literais, usados normalmente para definir tipos enumerados)	<Branco>, <Preto>

Tipos construídos - colecções

Descrição	Sintaxe	Exemplo de instância
Conjunto de elementos do tipo A	set of A	{1 , 2}
Sequência de elementos do tipo A	seq of A	[1, 2, 1]
Sequência não vazia	seq1 of A	
Mapeamento de elementos do tipo A para elementos do tipo B (função finita, conjunto de pares chave-valor)	map A to B	{ 0 |-> false, 1 |-> true }
Mapeamento injectivo (a chaves diferentes correspondem valores diferentes)	inmap A to B	

Outros tipos construídos

Descrição	Sintaxe	Exemplo de instância
Produto dos tipos A, B, … (instâncias são tuplos)	A * B * ...	mk_(0, false)
Record T com campos *a*, *b*, etc. de tipos *A*, *B*, etc. (*)	T :: a : A b : B ...	mk_T(0, false)
União dos tipos A, B, … (tipo A ou tipo B ou …)	A | B | ...	
Tipo **opcional** (admite **nil**)	[A]	

(*) Definições alternativas:

```
T :: a : A
     b :- B    -- campo com ":-" é ignorado na comparação de records

T :: A B       -- campos anónimos
```


Comparação de VDM++ com OCL

Tipo de dados composto	VDM++	OCL
Conjunto de elementos do tipo *A*	set of A	Set(A)
Conjunto admitindo repetidos		Bag(A)
Sequência de elementos do tipo *A*	seq of A	Sequence(A)
Sequência não vazia	seq1 of A	
Sequência sem repetidos		OrderedSet(A)
Mapeamento (função) de elementos do tipo *A* para elementos do tipo *B*	map A to B	
Mapeamento injectivo	inmap A to B	
Tuplo de tipo *T* com componentes *a*, *b*, ... de tipos *A*, *B*, ...	A * B * ... (anónimo) T :: a : A (c/nomes) b : B ...	Tuple(a : A, b : B, ...)
União (alternativa)	A | B | ...	

Strings

- Não está pré-definido o tipo string, mas pode ser facilmente definido como sequência de caracteres (seq of char)
- Todas as operações sobre sequências podem ser usadas com strings
- Strings literais podem ser indicadas com aspas
  - "eu sou" é equivalente a ['e', 'u', ' ', 's', 'o', 'u']

Exemplo de definição de tipos

```
class Pessoa
  types
    public String = seq of char;
    public Date :: year : nat
                  month: nat
                  day  : nat;
    public Sexo = <Masculino> | <Feminino>;
  instance variables
    private nome: String;
    private sexo: Sexo;
    private dataNascimento: Date;
    ...
end Pessoa
```

sequência

record

tipo enumerado (definido com *union* e *quote*)

atributo

tipo de dados

O tipo referência

- Referência para objecto de classe
- Permite modelar associações entre classes e trabalhar com objectos de classes
- Exemplo:

```
class Pessoa
  instance variables
     private conjuge : [Pessoa];
     private filhos : set of Pessoa;
...
```

Guarda referência para um objecto da classe Pessoa, ou nil

Guarda conjunto de 0 ou mais referências para objectos da classe Pessoa

Constantes simbólicas

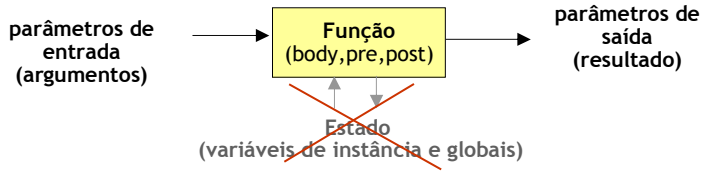
- São constantes às quais é dado um nome, por forma a tornar a especificação mais legível e fácil de alterar
- São declaradas na secção *values* com a sintaxe:

  [private | public | protected] *nome* [: *tipo*] = *valor*;

- Exemplo:

```
values
   public PI = 3.1417;
```

Funções



- Funções puras, sem efeitos laterais, convertem entradas em saídas
- Não têm acesso (seja para leitura ou alteração) ao estado do sistema representado pelas variáveis de instância
- São definidas na secção *functions*
- Podem ser *private* (por omissão), *public* ou *protected*
- Podem ser *static* (caso normal)
- Podem ter pré-condição, corpo (para definição explícita, paradigma funcional) e pós-condição (para definição implícita)

Funções - definição

- Estilo 1:

podem existir vários parâmetros de saída

```
f(a:A, b:B, …, z:Z) r1:R1, …, rn:Rn ==
   bodyexpr
pre preexpr(a,b,…,z)
post postexpr(a,b,…,z,r1,…,rn) ;
```

- Estilo 2:

```
f: A * B * … * Z -> R1 * R2 * … * Rn
f(a,b,…,z) ==
  bodyexpr
pre preexpr(a,b,…,z)
post postexpr(a,b,…,z,RESULT) ;
```

(valor simples ou tuplo)

Funções – partes da definição

- Corpo – definição explícita do(s) resultado(s) da função por uma expressão sem efeitos laterais
  - Paradigma funcional, executável (permite calcular o resultado)
  - Pode-se omitir: escrever "is not yet specified" ou omitir "==bodyexpr" no estilo 1
- Pré-condição (pre) - restrição nos valores dos argumentos que se deve verificar na chamada da função
  - Permite definir funções parciais (não definidas p/ alguns valores dos argumentos)
  - Pode ser omitida (mesmo que true)
  - A pré-condição de uma função f é também uma função chamada pre_f
- Pós-condição (post) – expressão booleana que relaciona resultado da função c/ argumentos (restrição a que deve obedecer o resultado)
  - Definição implícita da função (permite verificar mas não calcular o resultado)
  - Pode ser omitida (mesmo que true)
  - A pós-condição de uma função f é também uma função chamada post_f

Funções - exemplos

- Definição explícita (executável), função total

```
public static IsLeapYear(year: nat1) res : bool ==
  year mod 4 = 0 and year mod 100 <> 0 or year mod 400 = 0;
```

- Definição implícita (não executável), função parcial

```
public static sqrt(x: real) res : real
  pre x >= 0
  post res * res = x and res >= 0;
```

- Combinação de definição explícita com definição implícita

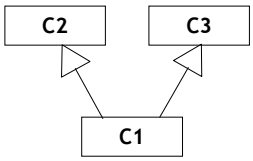
```
public static qsort(s: seq of nat) res: seq of nat ==
 cases s:
   []       -> [],
   [x]      -> [x],
   [x, y]   -> if x < y then [x, y] else [y, x],
   -^[x]^- -> qsort([y|y in set elems s & y<x])^[x]^qsort([y|y in set elems s & y>x])
 end
post IsSorted(res) and IsPermutation(res, s);
```

Herança

- Uma classe pode ter várias super-classes (herança múltipla)
- Sintaxe:

```
class C1 is subclass of C2, C3
  ...
end C1
```

- Semântica habitual
- Polimorfismo



Operadores e expressões

- Os já conhecidos de VDM-SL
  - Operadores aritméticos, lógicos e relacionais
  - Operadores sobre tipos construídos (colecções, records, tuplos, etc.)
  - Construção de colecções em compreensão e extensão
  - Expressões condicionais (if-then-else, cases)
  - Padrões
  - Quantificadores
  - Teste de pertença a tipo
- Alguns novos
  - Teste de pertença a classe
  - Criação de objectos
- Ver referência rápida e alguns exemplos em anexo

Aspectos sintácticos

- Comentários iniciam-se com " -- " e vão até ao fim da linha
- Distinção de minúsculas e maiúsculas (*case sensitive*)
- Acentos são suportados parcialmente, é preferível não usar
- Para referir um membro de instância (variável de instância ou operação) de um objecto, usa-se a notação habitual "objecto.membro"
- Para referir um membro estático (variável, operação ou função estática), tipo ou constante definido noutra classe, usa-se a notação "classe`membro", e não "classe.membro"
- Usa-se " nil " e não "null"
- Usa-se "self" e não "this"

Notação matemática ↔ Notação ASCII

·	&
×	*
≤	<=
≥	>=
≠	<>
$\xrightarrow{o}$	==>
→	->
⇒	=>
⇔	<=>

↦	|->
$\triangleq$	==
↑	**
†	++
⩒	munion
◁	<:
▷	:>
⩤	<-:
⩥	:->
⊂	psubset
⊆	subset
↷	^
⋂	dinter
⋃	dunion
$\mathcal{F}$	power
...-set	set of ...
...*	seq of ...
...+	seq1 of ...
$...\xrightarrow{m}...$	map ... to ...

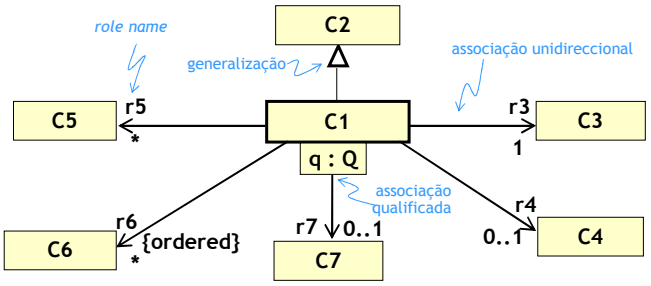
$...\overset{m}{\longleftrightarrow}...$	inmap ... to ...
μ	mu
$\mathbb{B}$	bool
$\mathbb{N}$	nat
$\mathbb{Z}$	int
$\mathbb{R}$	real
¬	not
∩	inter
∪	union
∈	in set
∉	not in set
∧	and
∨	or
∀	forall
∃	exists
∃!	exists1
λ	lambda
ι	iota
...-1	inverse ...

Exercício - VDM Tools

- Seguir o tutorial de utilização das VDM Tools apresentado em anexo, até à parte de testes, exclusive

Importação de diagramas de classes UML para VDM++

Mapeamento de relações entre classes

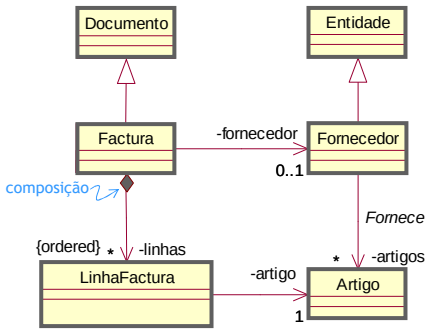



```
class C1 is subclass of C2
  instance variables
     r3 : C3;
     r4 : [C4];
     r5 : set of C5;
     r6 : seq of C6;
     r7 : map Q to C7;
end C1
```

seq1 se for para 1..*

inmap se for de 0..1

Exemplo

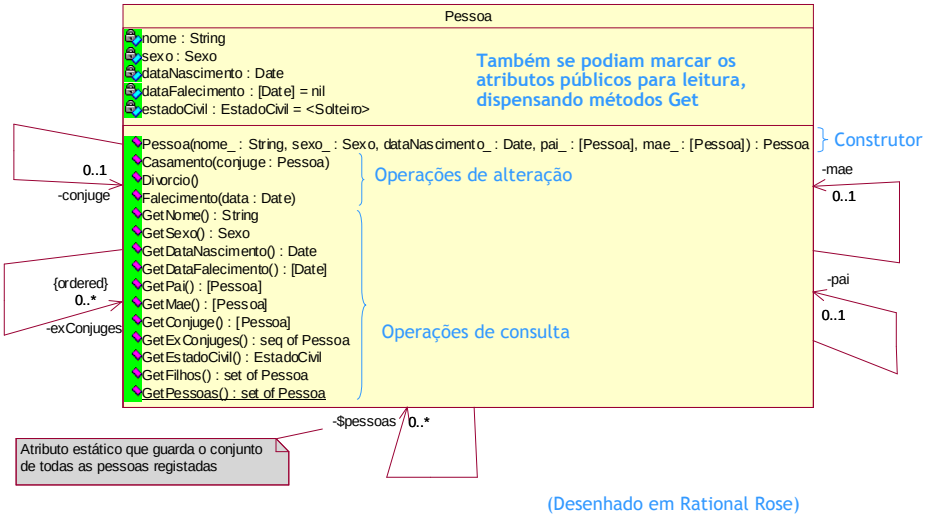



```
class Documento  end Documento;

class Entidade end Entidade;

class Artigo end Artigo;

class Fornecedor is subclass of Entidade
  instance variables
    private artigos: set of Artigo;
end Fornecedor;

class Factura is subclass of Documento
  instance variables
    private fornecedor: [Fornecedor];
    private linhas: seq of LinhaFactura
end Factura;

class LinhaFactura
  instance variables
    private artigo: Artigo;
end LinhaFactura;
```

Cuidados a ter na elaboração do diagrama de classes em UML

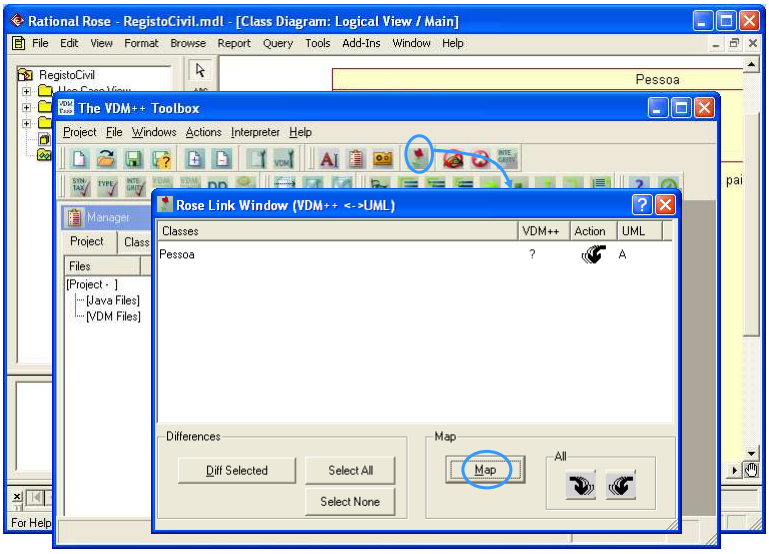
- Nomes de classes, atributos e operações: identificadores válidos, s/ espaços
  - Acentos e cedilhas funcionam mal em nomes de classes
- Indicar sentido de navegação nas associações
  - Associações são mapeadas p/referências entre objectos (como em linguagens OO)
  - Sentido da navegação indica que objectos guardam referências para que objectos
  - Se for necessário navegar (guardar referências) nos dois sentidos, criar duas associações navegáveis em sentidos opostos (c/ restrição adicional!)
- Indicar *role names* nos extremos navegáveis das associações
  - O *role name* serve para designar o objecto ou conjunto de objectos relacionados
- Os nomes dos parâmetros das operações de alteração de valores de atributos devem ser diferentes dos nomes dos atributos
  - Senão, no VDM++ não se consegue desambiguar a atribuição
- Usar tipos de dados da linguagem alvo (neste caso, VDM++)
- Atender a que em VDM++ (tal como nas linguagens de programação OO), não existe acesso automático ao conjunto de instâncias duma classe (como OCL)

Exemplo do Registo Civil



(Desenhado em Rational Rose)

Importação de Rational Rose para VDM++



Resultado da importação para VDM++ (1)

Ficheiro "Pessoa.rtf"

```
class Pessoa

instance variables
  private nome : String;
  private sexo : Sexo;
  private dataNascimento : Date;
  private dataFalecimento : [Date] := nil;
  private estadoCivil : EstadoCivil := <Solteiro>;
  private pai : [Pessoa];
  private mae : [Pessoa];
  private conjuge : [Pessoa];
  private exConjuges : seq of Pessoa;
  private static pessoas : set of Pessoa;
...
```

Mapeamento de atributos

Mapeamento de associações

Resultado da importação para VDM++ (2)

(após alteração de estilo, para ficar mais compacto)

Construtor

Operações de alteração

Operações de consulta directa de variáveis de instância

Outras consultas

```
...
operations
  public Pessoa(nome_ : String, sexo_ : Sexo, dataNascimento_ : Date,
                pai_, mae_ : [Pessoa]) res : Pessoa == is not yet specified;
  public Casamento(conjuge_ : Pessoa) == is not yet specified;
  public Divorcio() == is not yet specified;
  public Falecimento(data: Date) == is not yet specified;
  public GetNome() res : String == is not yet specified;
  public GetSexo() res : Sexo == is not yet specified;
  public GetDataNascimento() res : Date == is not yet specified;
  public GetDataFalecimento() res : [Date] == is not yet specified;
  public GetPai() res : [Pessoa] == is not yet specified;
  public GetMae() res : [Pessoa] == is not yet specified;
  public GetConjuge() res : [Pessoa] == is not yet specified;
  public GetExConjuges() res: seq of Pessoa == is not yet specified;
  public static GetPessoas () res: set of Pessoa == is not yet specified;
  public GetEstadoCivil() res: EstadoCivil == is not yet specified;
  public GetFilhos() res: set of Pessoa == is not yet specified;
end Pessoa
```

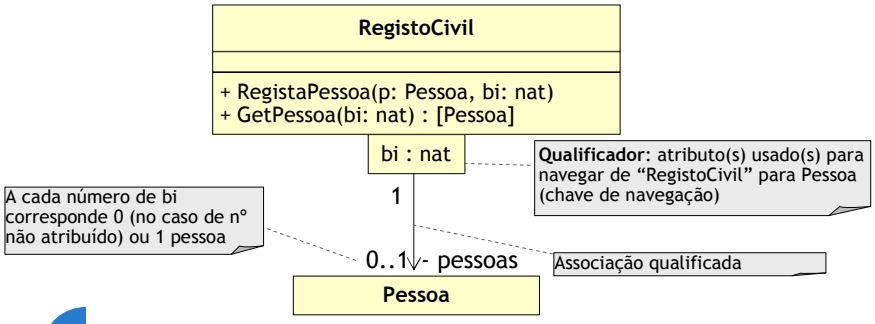
acrescentado

Acrescentar a definição de tipos de dados

```
class Pessoa

types
  public String = seq of char;
  public Date :: year  : nat1
                 month : nat1
                 day   : nat1;
  public Sexo = <Masculino> | <Feminino>;
  public EstadoCivil = <Solteiro> | <Casado> | <Divorciado> | <Viuvo> | <Falecido>;

instance variables
  ...

operations
  ...

end Pessoa
```

Já passa na verificação de sintaxe e tipos
(mas não é suficiente para executar)!

*Mapeamento de associações qualificadas



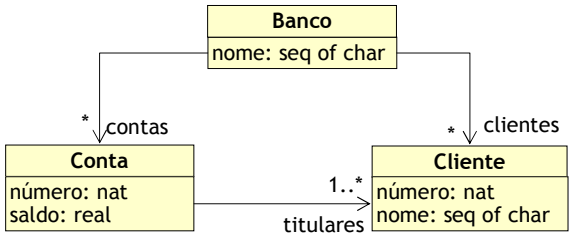

```
class RegistoCivil
  instance variables
    private pessoas : inmap nat to Pessoa :=  { |-> }  ;
  operations
    public RegistaPessoa(p : Pessoa, bi: nat) == is not yet specified ;
    public GetPessoa(bi: nat) res : [Pessoa] == is not yet specified;
end RegistoCivil
```


Que sentido(s) de navegação escolher?

- VDM++ não tem suporte nativo para associações bidireccionais nem mantém o conjunto de instâncias de uma classe
  - Tal como linguagens de programação OO Java, C++, C#, etc.
  - Mas ao contrário de UML, OCL, Executable UML, bases de dados relacionais, etc.
- Solução comum: objecto raiz que dá acesso aos restantes objectos do sistema, de forma hierárquica
  - Algumas navegações em sentido inverso
- Ver exemplo da Agenda Corporativa em anexo

Exercício - Contas bancárias

- Criar no Rational Rose o diagrama UML indicado abaixo e importar para as VDM Tools
- Acrescentar depois operações para criar contas e clientes e efectuar depósitos, levantamentos e consultas, e voltar a importar



Definição de invariantes

Invariantes de tipos

- A seguir à definição de um tipo pode-se definir um invariante, para restringir as instâncias válidas (valores válidos)
  inv *padrão* == *predicado*
  - *padrão* faz *match* com um valor do tipo em causa
  - *predicado* é a restrição a que o valor deve obedecer
- Normalmente o padrão é simplesmente uma variável, como em

```
types
  public Date :: year : nat1
                 month: nat1
                 day  : nat1
      inv d == d.month <= 12 and
               d.day <=  DaysOfMonth(d.year, d.month);
```

- Mas podem-se usar padrões mais complexos, por exemplo

```
inv mk_Date(y,m,d) == m <= 12 and d <=  DaysOfMonth(y, m);
```


Invariantes de estado

- Definem-se na secção "instance variables", a seguir à declaração das variáveis de instância, com a sintaxe
  inv *expressão_booleana_nas_variáveis_de_instância*;
- Restringem os valores válidos das variáveis de instância
- Em VDM++, os invariantes são verificados após cada atribuição
  - Atribuição a variável de instância da mesma classe do invariante!
- Também é possível agrupar várias atribuições num único bloco atómico, e verificar os invariantes só no final
  - Necessário p/ invariantes que relacionam diferentes var.s de instâncias
- São herdados por subclasses, que podem acrescentar outros
- A expressão de um invariante não deve ter efeitos laterais (pode invocar operações de consulta mas não de alteração de estado)

Tipos de invariantes comuns

- Restrição ao domínio (conj. de valores possíveis) de atributos
- Restrições de unicidade (chaves)
- Restrições relacionadas com ciclos nas associações
- Restrições temporais (com datas, horas, etc.)
- Restrições devidas a elementos derivados (calc. ou replic.)
- Regras de (condições para) existência (de valores ou objectos)
- Restrições de negócio genéricas
- Restrições idiomáticas (garantidas estruturalmente em UML mas não ao mapear para VDM++)

Tipos de invariantes comuns
Restrições ao domínio de atributos

A taxa de juros de um empréstimo é uma percentagem entre 0 e 100%.

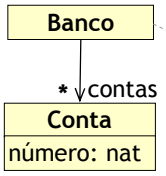
Empréstimo
taxaJuros: Percentagem

```
class Empréstimo
types
  Percentagem = real
    inv p == p >= 0 and p <= 100;
instance variables
  taxaJuros: Percentagem;
end Empréstimo
```

Normalmente é preferível definir por invariante de tipo!

Tipos de invariantes comuns
Restrições de unicidade (chaves)

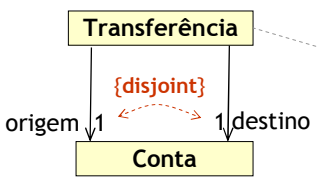
Um banco não pode ter duas contas com o mesmo número




```
class Banco
instance variables
  contas: set of Conta;

  inv not exists c1, c2 in set contas &
        c1 <> c2 and c1.número = c2.número;

...
```


Tipos de invariantes comuns
Restrições em ciclos nas associações: *disjoint*

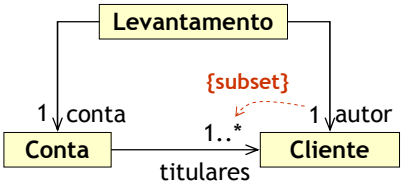
Uma transferência só pode ser efectuada entre contas diferentes




```
class Transferência
instance variables
  origem: Conta;
  destino: Conta;
  inv origem <> destino;
...
```

Tipos de invariantes comuns
Restrições em ciclos nas associações: *subset*

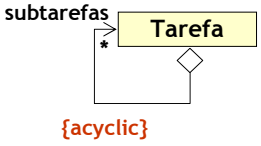
Um levantamento só pode ser efectuado por um dos titulares da conta




```
class Levantamento
instance variables
  conta: Conta;
  autor: Cliente;
  inv autor in set conta.titulares;
...
```


Tipos de invariantes comuns
Restrições em ciclos nas associações: *acyclic*

Um tarefa não pode ser subtarefa de si própria, directa ou indirectamente



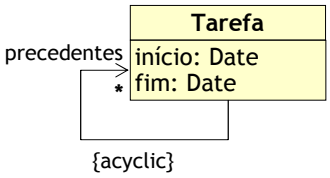
Definido por forma a não entrar em ciclo infinito no caso de existirem ciclos!

Como generalizar para reutilizar?

```
class Tarefa
instance variables
  subtarefas: set of Tarefa;
  inv self not in set fechoTransitivoSubTarefas();

operations
  fechoTransitivoSubTarefas() : set of Tarefa ==  (
      dcl fecho : set of Tarefa := subtarefas;
      dcl visitadas : set of Tarefa := {};
      while visitadas <> fecho do
          let t in set (fecho \ visitadas) in (
              fecho := fecho union t.subtarefas;
              visitadas := visitadas union {t}
          )
      return fecho
  );
...
```

Tipos de invariantes comuns
Restrições temporais

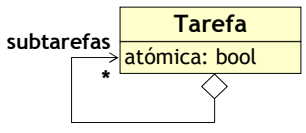
(1) Uma tarefa não pode terminar antes de começar
(2) Uma tarefa não pode começar antes das precedentes terminarem




```
class Tarefa
types
  Date = nat; -- YYYYMMDD
instance variables
  início: Date;
  fim: Date;
  precedentes: set of Tarefa;

  inv  fim >= início;
  inv  forall p in set precedentes &
         self.início >= p.fim;
...
```


Tipos de invariantes comuns
Regras de existência de objectos

Uma tarefa atómica não pode ter subtarefas




```
class Tarefa
instance variables
  atómica: bool;
  subtarefas: set of Tarefa;

  inv  atómica => subtarefas = {};
...
```

Tipos de invariantes comuns
Regras de existência de valores

Não se pode definir o fim efectivo de uma tarefa sem ter definido o seu início efectivo

multiplicidade de atributo

(UML)

ou

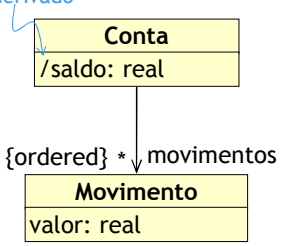
(VDM)

```
class Tarefa
instance variables
  inícioEfectivo: [Date];
  fimEfectivo: [Date];

  inv  fimEfectivo <> nil  =>
          inícioEfectivo <> nil;
...
```


Tipos de invariantes comuns
Restrições em elementos derivados: atributos

O saldo da conta é igual à soma dos valores dos movimentos desde a abertura da conta (negativos nos levantamentos)

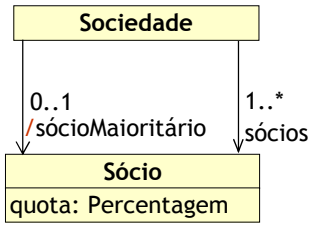
elemento derivado




```
class Conta
instance variables
  saldo: real;
  movimentos: seq of Movimento;

  inv total = sum ( {movimentos(i).valor &
                     i in set inds movimentos} );

functions
   sum(s: set of real) res : real ==
      if s = {} then 0
      else let x in set s in x + sum(s \ {x});
...
```

Tipos de invariantes comuns
Restrições em elementos derivados: associações

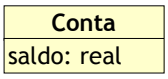
O sócio maioritário é o que tem uma quota superior a 50%




```
class Sociedade
instance variables
  sócios: set of Sócio;
  sócioMaioritário: [Sócio];

  inv sum({sócio.quota & sócio in sócios}) = 100;

  inv if exists1 s in set sócios & s.quota > 50
      then sócioMaioritário =
                iota s in set sócios & s.quota > 50
      else sócioMaioritário = nil;
...
```


Tipos de invariantes comuns
Regras de negócio genéricas

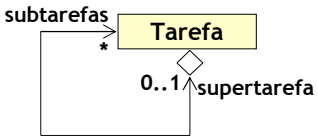
O saldo da conta não pode ser negativo




```
class Conta
instance variables
  saldo: real;

  inv saldo >= 0;

...
```

Tipos de invariantes comuns
Restrições idiomáticas (1)

Uma associação bidireccional é representada em VDM++ por duas associações unidireccionais, com restrições de integridade associadas



Ambos os invariantes são necessários (porquê?)

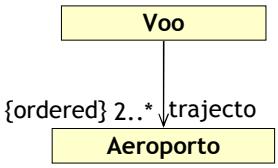
Pode ser visto como caso de ciclo em associações

```
class Tarefa
instance variables
  subtarefas: set of Tarefa;
  supertarefa: [Tarefa];

  inv forall t in set subtarefas &
        t.supertarefa = self;

  inv supertarefa <> nil =>
        self in set supertarefa.subtarefas;
```


Tipos de invariantes comuns
Restrições idiomáticas (2)

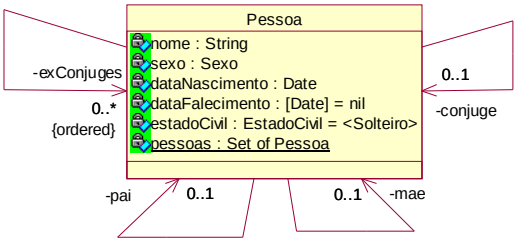
VDM++ não tem nativamente colecções ordenadas sem repetições (OrderedSet em OCL)

Restrições de multiplicidade podem originar invariantes




```
class Voo
instance variables
  trajecto: seq of Aeroporto;

  inv not exists i, j in set inds trajecto &
          i <> j and trajecto(i) = trajecto(j);

  inv len trajecto >= 2;
...
```

Exemplo do Registo Civil



- R1 – O pai tem de ser do sexo masculino
- R2 – A mãe tem de ser do sexo feminino
- R3 – Os cônjuges têm de ser de sexos opostos
- R4 – O falecimento tem de ser posterior ao nascimento
- R5 – Os pais têm de nascer antes dos filhos

Exemplo do Registo Civil: Formalização (1)

```
class Pessoa
 …
instance variables
…

 -- R1. O pai tem de ser do sexo masculino
 inv pai <> nil => pai.sexo = <Masculino>;

 -- R2. A mãe tem de ser do sexo feminino
 inv  mae <> nil => mae.sexo = <Feminino>;

 -- R3a. Os cônjuges têm de ser de sexos opostos
 inv conjuge <> nil => self.sexo <> conjuge.sexo;

 -- R3b. Os ex-cônjuges também têm de ser de sexos opostos
inv forall ex in set elems exConjuges & self.sexo <> ex.sexo;
```

Exemplo do Registo Civil: Formalização (2)

```
-- ***** Restrições temporais *****
-- R4. O falecimento não pode ser anterior ao nascimento
inv dataFalecimento <> nil =>
        not IsAfter(dataNascimento, dataFalecimento);
-- R5a. O pai tem de ter data de nascimento anterior ao filho
inv pai <> nil => IsAfter(dataNascimento, pai.dataNascimento);

-- R5b. A mãe tem de ter data de nascimento anterior ao filho
inv  mae <> nil => IsAfter(dataNascimento, mae.dataNascimento);

-- ****** Ciclos nas associações *****
-- Simetria de (ex)cônjuge: se A é (ex)cônjuge de B, então B é (ex)cônjuge de A
inv conjuge <> nil => conjuge.conjuge = self;
inv forall ex in set elems exConjuges & self in set elems ex.exConjuges;
```

Função auxiliar a definir p/ comparar datas

2 invariantes não suportados em VDM++!

Exemplo do Registo Civil: Formalização (3)

```
-- *** Regras de existência (consistência do estado civil) ***

-- A data de falecimento está definida sse o estado civil é "Falecido"
inv estadoCivil = <Falecido>   <=>  dataFalecimento <> nil;

-- O cônjuge está definido sse o estado civil é "Casado"
inv estadoCivil = <Casado>   <=>  conjuge <> nil;

-- Uma pessoa solteira não pode ter ex-cônjuges
inv estadoCivil = <Solteiro>   =>  exConjuges = [];

-- Uma pessoa divorciada ou viúva tem pelo menos um ex-cônjuge
inv estadoCivil in {<Divorciado>, <Viuvo>}   =>  exConjuges <> [];
...
operations
...
end Pessoa
```

A que classe associar cada invariante?

- Tanto em VDM++ como OCL, os invariantes têm de ser formalizados no contexto de uma classe
- No caso de invariantes que referem apenas uma classe, a decisão é trivial
- No caso de invariantes que envolvem mais do que uma classe, é uma decisão de "design" não trivial
  - classe onde a expressão é mais simples
  - classe onde se tem acesso a toda a informação
  - classe onde ocorrem operações que podem violar o invariante

Limitações de VDM++: invariantes inter-objecto

```
class A
  instance variables
    private x : nat;
    private b : B;
    inv x < b.GetY();
  operations
   public SetXY(newX, newY: nat) == (
      x := newX;
      b.SetY(newY)
   )
end A
```

```
class B
  instance variables
    private y : nat;
  operations
    public GetY() res: nat ==
       return y;
    public SetY(newY: nat) ==
       y := newY;
end B
```

1) Invariante é testado aqui (cedo de mais), não há maneira de atrasar verificação p/ fim do bloco!

2) Invariante não é testado aqui, pois está definido noutra classe!

Outras linguagens (OCL, Spec#, etc.) resolvem o 1º problema verificando invariantes só nos limites da chamada de métodos!

Exercício - Contas bancárias

- Continuar exercício das contas bancárias, acrescentando invariantes

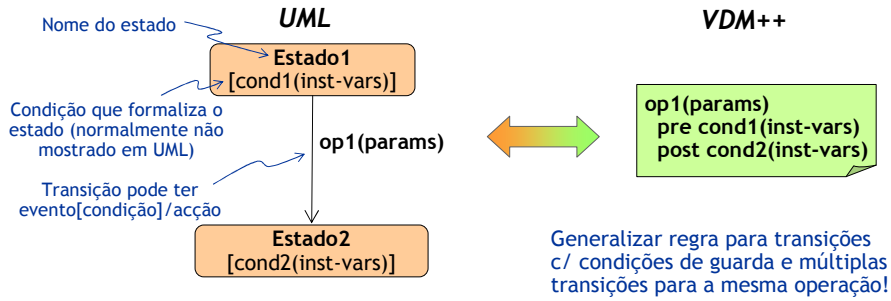
Definição de pré e pós-condições e relação com diagramas de estados UML

Pré e pós-condições de operações

- Pré-condição: restringe as condições de chamada (valores de argumentos e var.s de instância do objecto)
  - Correspondem em programação defensiva a validações efectuadas no início dos métodos (com possível lançamento de excepções)
- Pós-condição: formaliza o **efeito** da operação, através de condição que relaciona os valores finais das variáveis de instância e o valor retornado com os valores iniciais das variáveis de instância (indicados com ~) e os valores dos argumentos
- As pré e pós-condições do construtor, junto com valores por defeito das variáveis de instância, devem garantir o **estabelecimento dos invariantes**, entre outros efeitos
- As pré e pós-condições das operações de alteração, devem garantir a **preservação de invariantes** (assumindo que o objecto verifica os invariantes no início, também verifica no final), entre outros efeitos

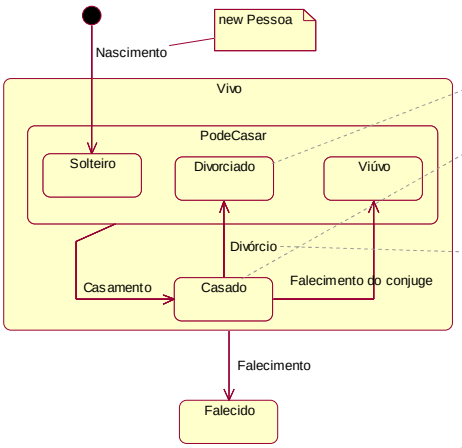
Relação com diagramas de estados UML

- Diagrama de estados é associado a uma classe e descreve o ciclo de vida e comportamento reactivo de cada objecto da classe (em resposta a eventos de chamada de operações ou outros a ver depois)
- Fornece restrições de integridade dinâmicas (transições válidas) para as pré e pós-condições das operações



Generalizar regra para transições c/ condições de guarda e múltiplas transições para a mesma operação!

Exemplo do Registo Civil



1º) Formalizar cada estado por uma condição nas variáveis de instância

estadoCivil = <Divorciado>

estadoCivil = <Casado>

2º) Obter trivialmente :

Divorcio()
pre estadoCivil = <Casado>
post estadoCivil = <Divorciado>;

3º) Completar, pois a operação pode ter outros efeitos que não estão detalhados no diagrama

Processo presentemente manual!

Exemplo do Registo Civil

Construtor

```
public Pessoa(nome0: String, sexo0: Sexo, dataNascimento0: Date,
              pai0, mae0: [Pessoa]) res: Pessoa ==
    is not yet specified
```

Variáveis de instância que a operação pode manipular

```
ext wr nome, sexo, dataNascimento, pai, mae, pessoas, estadoCivil
```

Garantir que argumentos não vão causar violação de invariantes

```
pre (pai0 <> nil =>
       pai0.sexo = <Masculino> and
       IsAfter(dataNascimento0, pai0.dataNascimento)) and
    (mae0 <> nil =>
       mae0.sexo = <Feminino> and
       IsAfter(dataNascimento0, mae0.dataNascimento))
```

Inicializar variáveis de instância c/ argumentos

Restrição dinâmica

Junta a conj. de instâncias

Retornar o próprio objecto

```
post nome = nome0 and
     sexo = sexo0 and
     dataNascimento = dataNascimento0 and
     pai = pai0 and
     mae = mae0 and
     estadoCivil = <Solteiro> and
     pessoas = pessoas~.union {self} and
     res = self;
```

Nas VDMTools, vai dar aqui o valor actual e não o antigo!

Exemplo do Registo Civil

```
public Casamento(conj: Pessoa) ==
    is not yet specified

ext wr estadoCivil, conjuge

pre estadoCivil in set {<Solteiro>, <Viuvo>, <Divorciado>} and
    conj.estadoCivil in set {<Solteiro>, <Viuvo>, <Divorciado>} and
    sexo <> conj.sexo

post estadoCivil = <Casado> and
     conjuge = conj and
     conj.estadoCivil = <Casado> and
     conj.conjuge = self;
```

Restrição dinâmica

Garantir invariante

Actualiza este objecto

Actualiza o outro objecto (conjuge)

A operação é chamada para uma das pessoas do casal, e trata de actualizar o estado das duas pessoas.
Dada a simetria, talvez ficasse melhor como operação estática com 2 argumentos.

Exemplo do Registo Civil

```
public Divorcio() ==
  is not yet specified

  ext wr estadoCivil, conjuge, exConjuges

  pre estadoCivil = <Casado>

  post estadoCivil = <Divorciado> and
       conjuge = nil and
       exConjuges = exConjuges~ ^ [conjuge~] and
       conjuge~.estadoCivil = <Divorciado> and
       conjuge~.conjuge = nil and
       self in set elems conjuge~.exConjuges;
       -- falha: conjuge~.exConjuges = conjuge~.exConjuges~ ^ [self];
```

Actualiza este objecto

Actualiza o outro objecto (conjuge)

A operação é chamada para uma das pessoas do casal, e trata de actualizar o estado das duas pessoas.
Dada a simetria, talvez ficasse melhor como operação estática com 2 argumentos.

Exemplo do Registo Civil

```
public Falecimento(data : Date) ==
    is not yet specified
 ext wr estadoCivil, dataFalecimento, conjuge, exConjuges
 pre data = nil and
     not IsAfter(dataNascimento, data)
 post estadoCivil = <Falecido> and
      dataFalecimento = data and
      if conjuge~ <> nil then (
        conjuge = nil and
        exConjuges = exConjuges~ ^ [conjuge~] and
        conjuge~.conjuge = nil and
        self in set elems conjuge~.exConjuges
        -- falha: conjuge~.exConjuges = conjuge~.exConjuges~ ^ [self]
      )
      else (
       exConjuges = exConjuges~ and
       conjuge = conjuge~
      );
```


Operações de consulta

- Também se podem especificar as operações de consulta com pós-condições

```
public GetNome() res : String ==
     is not yet specified
  ext rd nome
  post res = nome;
```

- Mas normalmente é mais útil escrever logo o corpo

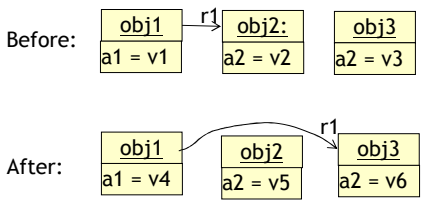
```
public GetNome() res : String ==
     return nome;
```

Limitações de VDM++: valores antigos

- Apenas é possível aceder ao valor inicial de variáveis de instância do próprio objecto (self)
- Não é possível a valores iniciais (antigos) de:
  - Variáveis de instância de objectos referenciados
  - Variáveis de instância herdadas de superclasses
  - Operações de consulta
  - Variáveis estáticas

Como OCL resolve essas limitações

- **propriedade@pre** – valor antigo da propriedade (atributo, associação ou operação de consulta), no início da execução da operação
- Pode-se usar "@pre" para aceder a valores antigos de propriedades de objectos referenciados



- **Existe equivalente em VDM++:**
  - obj1.a1 = v4
  - obj1.r1 = obj3
  - obj1.r1.a2 = v6
  - obj1.a1@pre = v1
  - obj1.r1@pre = obj2
  - obj1.r1@pre.a2 = v5
- **Não existe equivalente em VDM++:**
  - obj1.r1@pre.a2@pre = v2
  - obj1.r1.a2@pre = v3

* Propriedades das pre/pós-condições

- Satisfabilidade (existência de solução)
  - ∃ comb. de valores finais de variáveis de instância e valor de retorno satisfazendo a pós-condição, ∀ comb. de valores iniciais das variáveis de instância e argumentos obedecendo aos invariantes e à pré-condição
- Determinismo (unicidade de solução)
  - Sempre que os requisitos assim o indiquem, escrever uma pós-condição determinística (que admite uma única solução)
  - Mas, por exemplo, num problema de optimização, a pós-condição pode restringir as soluções admissíveis, sem chegar a impor uma solução única
- Respeito pelos invariantes
  - Se valores iniciais de var.s de instância e argumentos obedecerem aos invariantes e pré-condição, a pós-condição garante invariantes no final
  - Depois de garantir que todas as operações respeitam os invariantes, pode-se desactivar a sua verificação (mais pesada que verificação incremental de pré/pós-condições)

* Pré/pós-condições e herança

- Ao redefinir uma operação herdada da superclasse, não se deve violar o contracto (pré e pós-condição) estabelecido na super-classe
- A pré-condição pode ser enfraquecida (relaxada) na subclasse, mas não fortalecida (não pode ser mais restritiva)
  - qualquer chamada que se prometia ser válida na pré-condição da superclasse, deve continuar a ser aceite na pré-condição da subclasse
  - pre_op_superclass => pre_op_subclass
- A pós-condição pode ser fortalecida na subclasse, mas não enfraquecida
  - a operação na subclasse deve continuar a garantir os efeitos prometidos na superclasse, podendo acrescentar outros efeitos
  - post_op_subclass => post_op_superclass
- *Behavioral subtyping*

* Pré/pós-condições e herança

```
class Figura
 types
  public Ponto :: x : real
                  y : real;

 instance variables
  protected centro : Ponto;

 operations

  public Resize(factor: real) ==
    is subclass responsibility
    pre factor > 0.0
    post centro = centro~;

end Figura
```

```
class Circulo is subclass of Figura
 instance variables
  private raio : real;
  inv raio > 0;

 operations
  public Circulo(c: Ponto, r: real) res: Circulo
   == ( raio := r; centro := c; return self )
   pre r > 0;

  public Resize(factor: real) ==
   raio := raio * abs(factor)
   pre factor <> 0.0
   post centro = centro~ and
        raio = raio~ * abs(factor);
end Circulo
```

pre Figura`Resize(...) ⇒ pre Circulo`Resize(...)

post Figura`Resize(...) ⇐ post Circulo`Resize(...)

Exercício – Contas bancárias

- Continuar exercício das contas bancárias, acrescentando definição de pré-condições e pós-condições

Definição do corpo algorítmico de operações com instruções e acções

Instruções

- Para o modelo ser executável, é necessário escrever o corpo das operações, na forma de uma instrução ou bloco de instruções
- O corpo também é chamado "corpo algorítmico", pois, enquanto na pós-condição se especifica "o quê" (efeito), no corpo indica-se "como" (algoritmo)
- A linguagem VDM++ permite descrever e testar o algoritmo a um nível de abstracção elevado, refiná-lo até ao nível desejado, e gerar um programa executável em Java ou C++ com as VDM Tools
- Algumas instruções disponíveis:
  - Instrução de atribuição
  - Instruções "let" e "def"
  - Instruções de controlo de fluxo: "if", "cases", "for", "while"
  - Instrução "return"
  - Blocos e declaração de variáveis locais

Acções

- Acção: qualquer expressão (e.g., new) ou instrução (e.g., atribuição) que altera o estado do sistema, ou seja, que cria ou elimina objectos ou modifica o seu estado (ou o estado de variáveis estáticas)
- Criar objecto: new *nome-da-classe*(*parâmetros-constructor*)
- Eliminar objecto: automático, como em Java e C#
  - São automaticamente eliminados quando deixam de ser referenciados
  - O que podemos fazer explicitam/ é remover um objecto duma colecção ou desreferenciar atribuindo nil (obj_ref := nil)
  - Evita erros e simplifica as especificações
  - Em contrapartida, impede saber que instâncias existem de uma dada classe num dado momento (em OCL é ClassName.allInstances)
- Modificar estado de objecto: ver operador de atribuição

Blocos e declaração de variáveis

```
(
   dcl id1 : tipo1 [:= expr1],  id2 : tipo2 [:= expr2], …;
   dcl … ;
   …
   instrução1;
   instrução2;
   …
)
```

- Um bloco tem de ter pelo menos uma instrução
- Variáveis só podem ser declaradas no início do bloco
- Última instrução não precisa de “;”
- A 1ª instrução que retornar um valor (mesmo sem “return”, basta chamar uma operação que devolva um valor) faz terminar o bloco

Atribuição

- *designador_de_estado := expressão*
  - Nome de variável
    - Variável de instância do objecto em causa
    - Variável estática (**static**)
    - Variável local da operação (declarada com **dcl**)
  - Parte de variável do tipo *map*, *seq* ou *record*
    - map_var(chave) := valor
    - seq_var(índice) := valor
    - record_var.campo := valor

Não se pode fazer *object_reference.instance_variable* := *expr* (mesmo que a variável de instância seja pública)!

Um identificador introduzido com let , forall, etc. não é uma variável neste sentido

Atribuição múltipla

- atomic (*sd1 := exp1*; *sd2 := exp2*; …)
- Avalia primeiro todas as expressões do lado direito, e só depois atribui (em simultâneo) os valores resultantes às variáveis do lado esquerdo!
- Só verifica invariantes no final das várias atribuições (senão, verificaria invariantes após cada atribuição)


Útil na presença de invariantes que envolvem mais do que uma variável de instância (do mesmo objecto)


Não resolve o problema de invariantes *inter-objecto*, isto é, que envolvem múltiplos objectos (porquê?)

Atribuição múltipla - exemplo

```
instance variables
  private quantidade : real;
  private precoUnitario : real;
  private precoTotal : real;
  inv precoTotal = quantidade * precoUnitario;
operations
```

```
public SetQuantidade(q: real) ==
  (quantidade := q;  precoTotal := precoUnitario * q);
```

Quebra invariante após 1ª atribuição

```
public SetQuantidade(q: real) ==
  atomic(quantidade := q; precoTotal := precoUnitario * q);
```

```
public SetQuantidade(q: real) ==
  atomic(quantidade:=q; precoTotal:=precoUnitario * quantidade);
```

Errado: usa valor antigo da variável

Instruções “let” e “def”

let *definição1*, *definição2*, … in *instrução*

let *identificador* in set *conjunto* [be st *condição*] in *instrução*

def *definição1*, *definição2*, … in *instrução*

- Têm a mesma forma que expressões “let” e “def”, com *instrução* em vez de *expressão* na parte de “in”
- Usar “def” em vez de “let”, quando na parte de definições são invocadas operações que alteram estado
- Identificadores introduzidos na parte de definições não são variáveis que possam mudar de valor (não podem aparecer do lado esquerdo de atribuições)!

Instruções condicionais "if" e "cases"

- if *condição* then *instrução1* [else *instrução2*]
- cases *expressão*:
  padrão11, *padrão12*, ..., *padrão1N* -> *instrução1*,
  ... -> ...,
  padrãoM1, *padrãoM2*, ..., *padrãoMN* -> *instruçãoM*,
  others -> *instruçãoM1*
  end

- Têm a mesma forma que as expressões "if" e "cases", com instruções em vez de expressões
- Na instrução "if", a parte de "else" é opcional

Ciclos "for" e "while"

Instrução	Descrição
while *condição* do *instrução*	Ciclo "while" tradicional
for *contador* = *N1* to *N2* [by *N3*] do *instrução*	Ciclo "for" tradicional, com inteiros. *Contador* não tem de ser declarado previamente.
for all *padrão* in set *conjunto* do *instrução*	Normalmente o padrão é simplesmente um identificador. Percorre os elementos do conjunto por uma ordem arbitrária. Não confundir com quantificador existencial "forall".
for *padrão* in *sequência* do *instrução*	Normalmente o padrão é simplesmente um identificador. Percorre os elementos da sequência por ordem.

Instrução "return"

- return
  - Usado para terminar operações que não retornam qualquer valor
- return *expressão*
  - Usado para terminar operações que retornam um valor

⚠ Cuidado com return implícito: a 1ª instrução que retornar um valor (basta chamar operação que devolve valor) faz terminar o bloco

Exemplo do Registo Civil (1)

```
public Pessoa(nome0: String, sexo0: Sexo, dataNasc0: Date,
              pai0, mae0: [Pessoa]) res: Pessoa ==
(
  atomic(
    nome := nome0;
    sexo := sexo0;
    dataNascimento := dataNasc0;
    pai := pai0;
    mae := mae0;
    pessoas := pessoas union {self};
  );
  return self
)
ext wr nome, sexo, dataNascimento, pai, mae, pessoas
pre  ...
post nome = nome0 and sexo = sexo0 and dataNascimento = dataNasc0
     and pai = pai0 and mae = mae0 and
     pessoas = pessoas~ union {self} and
     res = self;
```

Para só verificar invariantes depois de inicializar todas as variáveis!

Exemplo do Registo Civil (2)

```
public Casamento(conj: Pessoa) ==
  (
    self.SetCasado(conj);
    conj.SetCasado(self)
  )
  ext wr estadoCivil, conjuge
  pre ...
  post estadoCivil = <Casado> and  conjuge = conj and
       conj.estadoCivil = <Casado> and conj.conjuge = self;

-- operação auxiliar interna
private SetCasado(conj: Pessoa) ==
   atomic (
        conjuge := conj;
        estadoCivil := <Casado>
   );
```

Tem de usar operação auxiliar por limitação do operador de atribuição

Ver restantes em RegistoCivil.zip

Escrever ou não a pós-condição? (em modelos executáveis)

- Se queremos obter um modelo executável (com corpo de operações), não há benefício em escrever pós-condições trivialmente semelhantes ao corpo
  - Ou o corpo podia ser gerado automaticamente da pós-condição …
- Mas, se o corpo é muito mais complexo que a pós-condição, pode ser vantajoso escrever a pós-condição
  - O corpo permite especificar o **algoritmo** a seguir na realização da operação
  - A pós-condição permite especificar o **objectivo** e **verificar** o resultado
  - Ver exemplo a seguir e problema da colocação de professores (em anexo)
- Outras vezes, a pós-condição pode ser usada para especificar algumas restrições a que deve obedecer o resultado, sem o fixar completamente
  - Caso típico de problemas de optimização, como na colocação de professores
- Em alguns casos, não é mesmo possível especificar o efeito pretendido através de pós-condições, logo é importante o corpo
  - Ver mais tarde caso de *callbacks* e *event listeners*

Ordenação topológica (1/2)

```
-- Operação de ordenação topológica dos vértices dum grafo dirigido.
-- Grafo representado por mapeamento de vértices para sucessores.
-- Algoritmo descrito em D. Knuth, The Art of Computing Programming, Vol. 1.
TopologicalSort : map Vertex to set of Vertex ==> seq of Vertex
TopologicalSort(Succ) == (
  dcl indegree: map Vertex to nat; -- nº de predecessores por ordenar
  dcl S : set of Vertex;  -- vértices por ordenar com indegree = 0
  dcl R : seq of Vertex := []; -- resultado da ordenação

  -- cálculo de indegree e inicialização de S
  indegree := {v |-> 0 | v in set dom Succ};
  for all v in set dom Succ do
     for all w in set Succ(v) do
        indegree(w) := indegree(w) + 1;
  S := {v | v in set dom Succ & indegree(v) = 0};
```


Ordenação topológica (2/2)

```
 -- cálculo da ordem topológica (R)
 while S <> {} do
    let v in set S in (
       S := S \ {v};
       R := R ^ [v];
       for all w in set Succ(v) do (
          indegree(w) := indegree(w) - 1;
          if indegree(w) = 0 then S := S union {w}
       )
    );
 return R
)
pre ((dunion rng Succ) subset (dom Succ))  and IsAcyclic(Succ)
post (elems RESULT = dom Succ)  and HasNoDuplicates(RESULT)
 and (forall i, j in set inds RESULT & i <= j => RESULT(i) not in set Succ(RESULT(j)));
```

Algoritmo pode ser implementado em tempo O(nº de vértices + nº de arestas).

Na presença de ciclos (violando a 2ª pré-condição), o algoritmo dá uma sequência parcial (violando a 1ª pós-condição)

Modelo executável pode estar mais perto da especificação ou da implementação!

- Definição implícita não executável:

```
public static sort(s: seq of nat) res: seq of nat ==
  post IsSorted(res) and IsPermutation(res, s);
```

- Definição explícita executável, baseada na definição implícita:

```
public static sort(s: seq of nat) res: seq of nat ==
  iota t in set Permutations(s) & IsSorted(t); -- iota: seleccção
```

- Definição explícita executável, segundo algoritmo "quick sort":

```
public static qsort(s: seq of nat) res: seq of nat ==
 cases s:
   []       -> [],
   [x]      -> [x],
   [x, y]   -> if x < y then [x, y] else [y, x],
   -^[x]^- -> qsort([y|y in set elems s & y<x])^[x]^qsort([y|y in set elems s & y>x])
 end;
```

Refinamento

Exercício – Contas bancárias

- Continuar exercício das contas bancárias, acrescentando corpo das operações
- Efectuar alguns testes usando o interpretador de VDM++

Teste da especificação

Teste da especificação

- Uma especificação bem construída já tem verificações *built-in*
  - Invariantes, pré/pós-condições, outras asserções (invariantes de ciclos, etc.)
- Mas deve ser exercitada de forma repetível com testes automatizados
  - O objectivo é descobrir erros e ganhar confiança na correcção da especificação
  - Mais tarde, os mesmos testes podem ser aplicados à implementação
- Testar com entradas válidas
  - Exercitar toda as partes da especificação (medir cobertura com VDMTools)
  - Usar asserções para verificar valores devolvidos e estados finais
  - (Op) Derivar testes a partir de máquinas de estados (teste baseado em estados)
  - (Op) Derivar testes a partir de cenários de utilização (teste baseado em cenários)
  - (Op) Derivar testes de especificações axiomáticas (teste baseado em axiomas)
- Testar com entradas inválidas
  - Quebrar todos os invariantes e pré-condições, para verificar que funcionam …

Suporte para teste nas VDM Tools

- Especificação pode ser testada interactivamente com interpretador de VDM++, ou com base em casos de teste pré-definidos
- Pode-se activar a verificação automática de invariantes, pré-condições e pós-condições
- Para obter informação de cobertura de testes, é necessário definir pelo menos um *script* de teste
  - Cada *script* de teste tsk é especificado por dois ficheiros:
    - ficheiro tsk.arg – com o comando a executar pelo interpretador
    - ficheiro tsk.arg.exp – com o resultado esperado da execução do comando
  - VDMTools dão informação dos testes que sucederam e que falharam
  - *Pretty printer* "pinta" as partes da especificação que foram de facto executadas e gera tabelas com % de cobertura e número de chamadas

Simulação de asserções

Utilização

```
class TestPessoa is subclass of Test
   operations
      public TestNome() == (
         dcl j : Pessoa := new Pessoa("João", …);
         Assert( j.GetNome() = "João")
      )
end TestPessoa
```

Definição

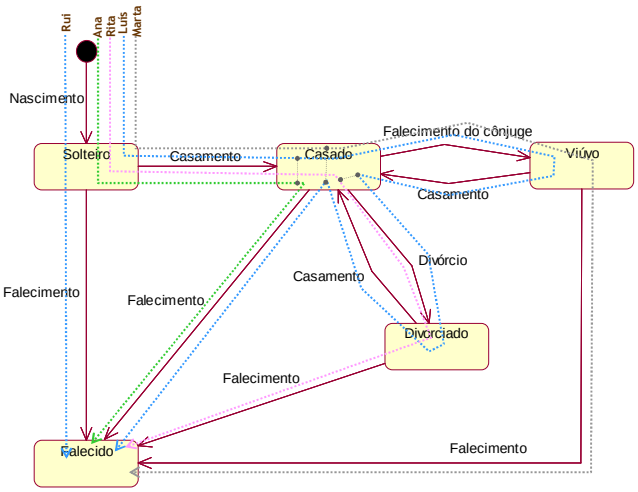
```
class Test
   operations
      protected Assert : bool ==> ()
         Assert(a) == return
      pre a
end Test
```

Violação de asserção é reduzida a violação de pré-condição (activar verificação de pré-condições nas VDMTools)

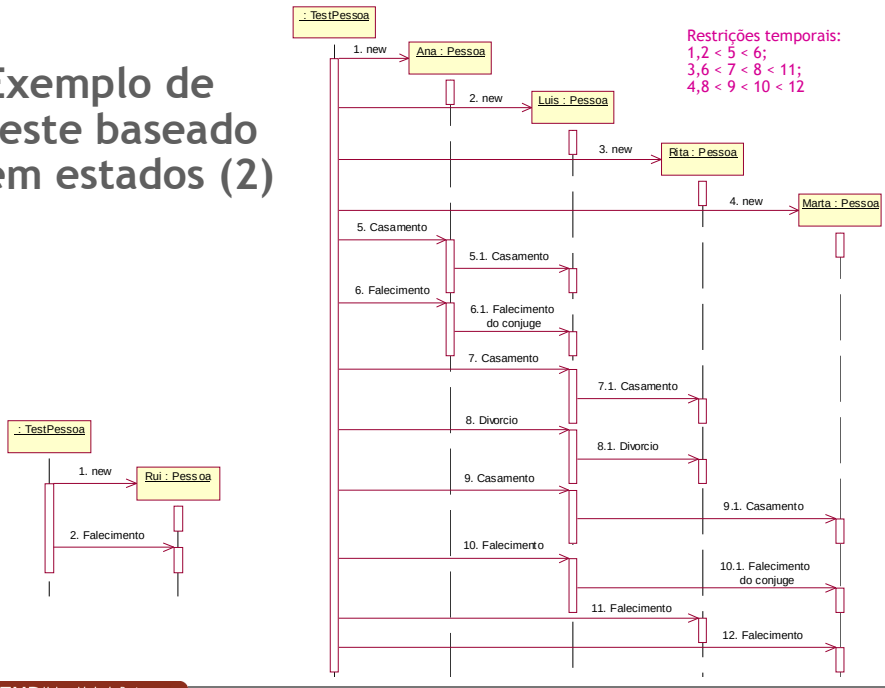
Teste baseado em estados

- Modelar comportamento (ciclo de vida) de tipos de objectos interessantes por diagramas de estados UML
- Determinar sequências de teste que cobrem (pelo menos) todos os estados e transições nos diagramas de estados
  - P/ testes mais completos, expandir elem.s compostos (estados, guardas, eventos)
  - Quando não há interacções entre objectos, uma seq. de teste é um caminho no diagrama de estados partindo do estado inicial (uma vida possível de um objecto)
  - Como um diagrama de estados UML representa uma máquina de estados estendida (com variáveis de estado adicionais), nem todos os caminhos no diagrama representam necessariamente comportamentos válidos
  - Quando há interacções entre objectos, uma sequência de teste é uma história de execução de um conjunto de objectos interdependentes (em que a vida de cada objecto corresponde a um caminho válido no diagrama de estados respectivo)
- (Opc) Representar sequências de teste por diagramas de sequência UML
- Converter para VDM++ e completar com verificação de resultados esperados
- Testar também eventos não válidos (violação de pré-condições)

Exemplo de teste baseado em estados (1)



Exemplo de teste baseado em estados (2)



Exemplo de teste baseado em estados (3)

```
public TestSingle() == (
  -- setup
   dcl r: Pessoa;
   dcl d1: DateUtils`Date := mk_DateUtils`Date(1980, 1, 31);
   dcl d2: DateUtils`Date := mk_DateUtils`Date(2022, 3, 1);

   -- exercitar
   r := new Pessoa("Rui", <Masculino>, d1, nil, nil);

   -- verificar
   Assert(r.GetEstadoCivil() = <Solteiro>);
   Assert(r.GetDataNascimento() = d1);

   -- exercitar
   r.Falecimento(d2);

   -- verificar
   Assert(r.GetEstadoCivil() = <Falecido>);
   Assert(r.GetDataFalecimento() = d2)
);
```

Exemplo de teste baseado em estados (4)

```
public TestCouples() == (
  dcl d1: DateUtils`Date := mk_DateUtils`Date(1960, 1, 1); -- nascimento
  dcl d3: DateUtils`Date := mk_DateUtils`Date(2000, 1, 1); -- falec. conjuge
  …
  dcl a: Pessoa := new Pessoa("Ana", <Feminino>, d1, nil, nil);
  dcl l: Pessoa := new Pessoa("Luis", <Masculino>, d1, nil, nil);
  …
  a.Casamento(l);
  Assert(a.GetEstadoCivil() = <Casado>);
  Assert(a.GetConjuge() = l);
  Assert(l.GetEstadoCivil() = <Casado>);
  Assert(l.GetConjuge() = a);

  a.Falecimento(d3);
  Assert(a.GetEstadoCivil() = <Falecido>);
  Assert(a.GetConjuge() = nil);
  Assert(a.GetExConjuges() = [l]);
  Assert(a.GetDataFalecimento() = d3);
  Assert(l.GetEstadoCivil() = <Viuvo>);
  Assert(l.GetConjuge() = nil);
  Assert(l.GetExConjuges() = [a]);
  …
```

Quantos asserts vale a pena fazer?

Teste baseado em cenários

- Cenários de utilização são também bons cenários de teste
  - Cenários normais
  - Cenários alternativas / excepcionais
- Cenários de utilização/teste podem ser representados por diagramas de sequência UML
- Cenários de utilização/teste podem ser formalizados por métodos de teste parametrizados em VDM++
  - Devem ser o mais genéricos possível
  - Podem ter pré/pós condições e asserções
- Instanciar depois os cenários com valores de teste concretos

Exemplo de teste baseado em cenários

```
class ATMTest is subclass of Test
 public SuccessfulWithdrawal(m: ATM, c: Card, p: Pocket, pin: nat, value: nat1) == (
  m.insertCard(c);
  Assert(m.getStatus() = <EnterPin>);
  m.enterPin(pin);
  Assert(m.getStatus() = <SelectOperation>);
  m.selectOperation(<Withdrawal>);
  Assert(m.getStatus() = <EnterAmount>);
  m.enterAmount(value);
  p.add(m.pickMoney());
  m.removeCard()
 )
 pre m.getStatus() = <InsertCard>
  and pin = c.getPin() and value <= c.getAccount().getBalance()
  and m.hasStock(value) and value <= m.getWithdrawalLimit()
 post c.getAccount().getBalance() = c.getAccount().getBalance()~ - value
  and m.getBalance() = m.getBalance()~ - value
  and p.getBalance() = p.getBalance()~ + value
  and m.getStatus() = <InsertCard>;
```

```
public TestWithdrawalLimit() == (
  dcl m : ATM := new ATM({10 -> 5, 20 -> 5});
  dcl a : Account := new Account(150);
  dcl c : Card := new Card(a, 1111);
  dcl p : Pockect := new Pocket({|->});
  m.SetWithdrawalLimit(150);
  SuccessfulWithdrawal(m, c, p, 1111, 150)
 )
end ATMTest
```


Teste baseado em axiomas

- Especificações axiomáticas (em OBJ, etc.) e testes partilham o facto de não usarem um modelo do estado interno do objecto (que normalmente está escondido)
- Tal como nos cenários, axiomas podem ser formalizados por métodos de teste parametrizados em VDM++
  - Válidos para quaisquer valores dos parâmetros que obedeçam à pré-condição
- Instanciar depois com valores de teste concretos

Exemplo de teste baseado em axiomas

Axiomas

Casos de teste

```
class StackTest  is subclass of Test

  -- Top(Push(s,x)) = x
  public PushTop(s : Stack , x: int) == (
     s.Push(x);
     Assert(s.Top() = x)
  );

  -- Pop(Push(s, x)) = s
  public PushPop(s: Stack, x: int) == (
     Stack spre = s.Clone();
     s.Push(x);
     s.Pop();
     Assert(s.Equals(spre))
  );

  public TestPushTop() == ( PushTop(new Stack(), 1) );
  public TestPushPop() == ( PushPop(new Stack(), 1) )
end StackTest
```


Test-Driven Development com VDM++

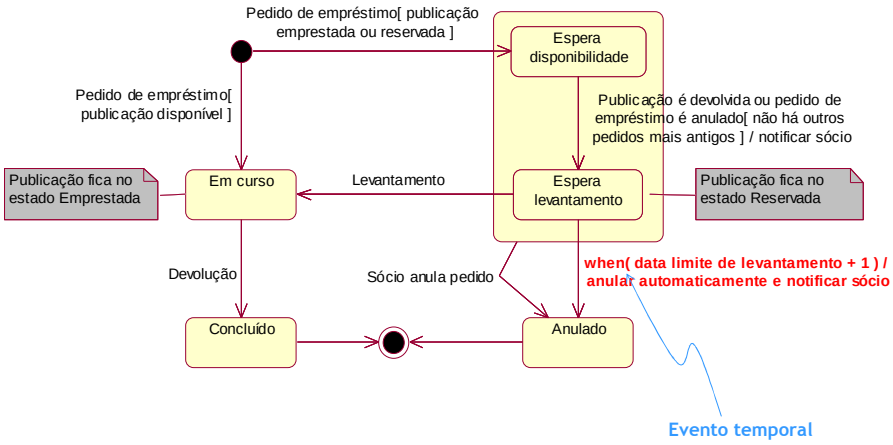
- Princípios:
  - Escrever os testes antes do objecto dos testes (em cada iteração)
  - Desenvolver por pequenas iterações
  - Automatizar os testes
  - "Refabricar" (*refactor*) para remover duplicação de código
- Vantagens de TDD:
  - Garantir qualidade dos testes
  - Pensar em casos particulares antes de pensar em casos gerais
    - casos de teste são especificações parciais
  - Sistemas complexos que funcionam resultam da evolução de sistemas mais simples que funcionam

Exercícios

- Seguir a parte final (sobre testes) do tutorial de utilização das VDM Tools (problema da Stack)
- Escrever testes para o problema da contas bancárias

Modelação de eventos temporais

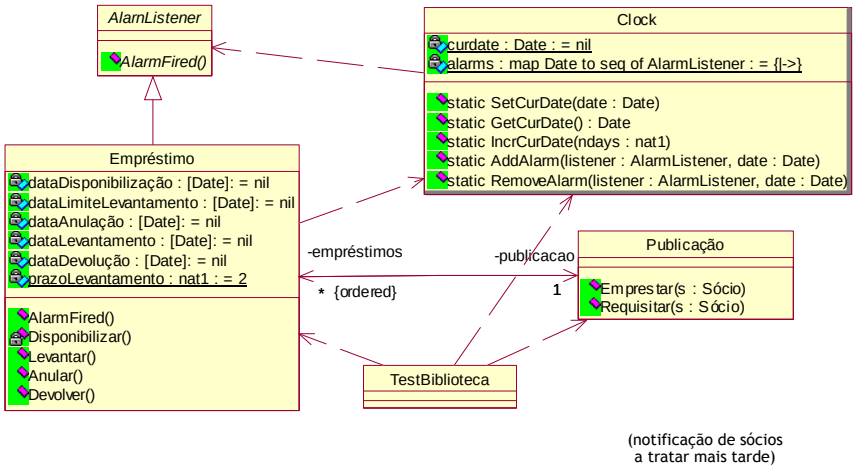
Exemplo de ciclo de vida de um empréstimo numa biblioteca



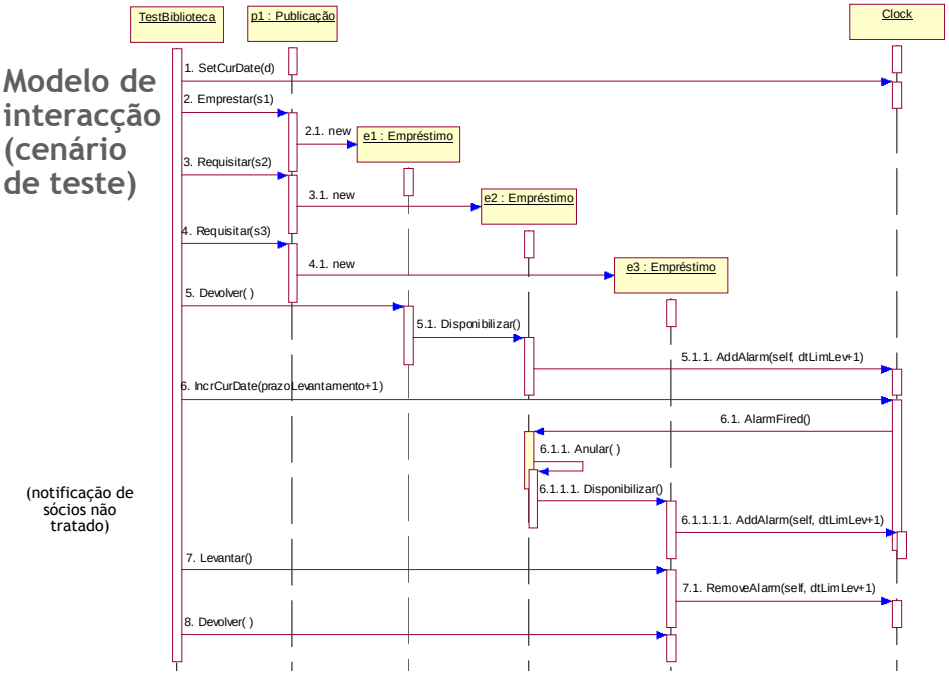
Modelação de eventos temporais em VDM++

- VDM++ não tem suporte nativo para eventos temporais
- Eventos temporais podem ser gerados por um relógio com alarme
- O padrão de desenho “Observer” pode ser usado na comunicação entre o relógio e os seus “clientes”
- Para efeito de modelação e teste, não é necessário usar a data do sistema, basta uma data simulada comandada pelo código de teste
- Para simplificar os testes, vamos considerar que o relógio funciona de forma síncrona com o resto do sistema
- Posteriormente, o modelo poderia ser refinado para o caso assíncrono, em que se teriam de tratar problemas de concorrência

Modelo de estrutura estática



Modelo de interacção (cenário de teste)



Classe "Clock" (1)

- Modelação do estado (a vermelho tem a ver com alarmes):

```
class Clock
 types
   public Date = DateUtils`Date;
 instance variables
   private static curdate : Date := DateUtils`MakeDate(2000,1,1);
   private static alarms : map Date to seq of AlarmListener := { |-> };
   private static changing_curdate : bool := false;

   -- não podem existir alarmes para datas passadas, nem para a data actual (a não
   -- ser transitoriamente enquanto se processam os alarmes após mudar a data)
   inv forall d in set dom alarms &
         if changing_curdate then d >= curdate else d > curdate;

   -- não podem existir "listeners" duplicados para a mesma data
   inv forall d in set dom alarms &
         not SeqUtils`HasDuplicates[AlarmListener](alarms(d));
```


Classe "Clock" (2)

- Definição de operações sobre a data corrente:

```
...
 operations
   public static GetCurDate() res : Date ==
     return curdate;

   public static SetCurDate(date : Date) ==
   (
     while curdate < date do
       IncrCurDate()
   )
   pre date >= curdate and not changing_curdate;

   public static IncrCurDate(n : nat1) ==
   (
     for i = 1 to n do
       IncrCurDate()
   )
   pre not changing_curdate;
...
```

Para não comprometer o processamento de alarmes!

Classe "Clock" (3)

- Definição de operação auxiliar que incrementa a data corrente e dispara os alarmes, ilustrando também limitações de pós-condições com *callbacks*

```
...
 private static IncrCurDate() ==
 (
   changing_curdate := true;
   curdate := DateUtils`NextDate(curdate);
   while curdate in set dom alarms do (
     dcl listener : AlarmListener := hd alarms(curdate);
     RemoveAlarm(listener, curdate);
     listener.AlarmFired()
   );
   changing_curdate := false
 )
 ext wr curdate, alarms, changing_curdate
 pre not changing_curdate -- não é reentrante!
 post curdate = DateUtils`NextDate(curdate~) and curdate not in set dom alarms
       and not changing_curdate;
  -- pós-condição não formalizável: os listeners registados para a nova data (e
  -- não removidos até chegar a sua vez), têm de ter sido chamados (com a data
  -- já actualizada) pela ordem por que se registaram
...
```


Classe "Clock" (4)

- Definição de operações para registar (adicionar) e remover alarmes (pós-condições omitidas):

```
...
 public static AddAlarm(listener : AlarmListener, date : Date) ==
 (
    if date in set dom alarms then  alarms(date) := alarms(date) ^ [listener]
    else alarms := alarms munion { date |-> [listener] }
 )
 pre date > curdate and not ExistsAlarm(listener, date);

 public static RemoveAlarm(listener : AlarmListener, date : Date) ==
 (
    alarms(date) := SeqUtils`Remove[AlarmListener](listener, alarms(date));
    if alarms(date) = [] then alarms := {date} <-: alarms
  )
  pre ExistsAlarm(listener, date);

  public static ExistsAlarm(listener : AlarmListener, date : Date) res: bool ==
    return date in set dom alarms   and   listener in set elems alarms(date);

end Clock
```

Classe "AlarmListener"

```
class AlarmListener

  operations

    public AlarmFired() ==
      is subclass responsibility;

end AlarmListener
```


Classe "Empréstimo" - Registar alarme

```
class Empréstimo is subclass of AlarmListener

…

 -- Chamado internamente aquando da devolução ou anulação doutro empréstimo
 private Disponibilizar() ==
 (
   dataDisponibilização := Clock`GetCurDate();
   dataLimiteLevantamento := DateUtils`DateAdd(dataDisponibilização,
                                               prazoLevantamento);
   sócio.Notificar(<PodeLevantar>, publicação, dataLimiteLevantamento);
   Clock`AddAlarm(self, DateUtils`NextDate(dataLimiteLevantamento))
 )
 pre GetEstado() = <EsperaDisponibilidade>
 post GetEstado() = <EsperaLevantamento>;
```

Classe “Empréstimo” - Reagir a alarme

```
-- Chamado de Clock quando o alarme dispara
-- (nota: é "public“ porque herda de “public”)
public AlarmFired() ==
(
  dataAnulação := Clock`GetCurDate();
  sócio.Notificar(<ExpirouPrazoLevantamento>, publicação,
                  dataLimiteLevantamento);
  def e2  = publicação.GetPróximoPedido() in if e2 <> nil then e2.Disponibilizar()
)
pre GetEstado() = <EsperaLevantamento>
post GetEstado() = <Anulado>;
```


Classe “Empréstimo” - Remover alarme

```
-- O sócio anula o pedido
public Anular() ==   (
  if GetEstado() = <EsperaLevantamento> then  (
     dataAnulação := Clock`GetCurDate();  -- altera estado
     Clock`RemoveAlarm(self, DateUtils`NextDate(dataLimiteLevantamento));
     def e2  = publicação.GetPróximoPedido() in if e2 <> nil then e2.Disponibilizar()
  )
 else
     dataAnulação := Clock`GetCurDate()
)
pre GetEstado() in set {<EsperaDisponibilidade>, <EsperaLevantamento>}
post GetEstado() = <Anulado>;
```

```
-- O sócio levanta a publicação pedida
public Levantar() == (
  dataLevantamento := Clock`GetCurDate();
  Clock`RemoveAlarm(self, DateUtils`NextDate(dataLimiteLevantamento))
)
pre GetEstado() = <EsperaLevantamento>
post GetEstado() = <EmCurso>;
```

Definição de pós-condições com *callbacks*

- No caso de operações c/ *callbacks* (e.g., invocação de *event listeners* ou *event handlers*), quem escreve a operação não sabe o efeito resultante no estado do sistema
- Nestes casos, o efeito pretendido é mais procedimental (e.g., invocar todos os "event listeners" registados, por ordem de registo)
- OCL permite exprimir efeitos de operações que incluem envio de mensagens para outros objectos com consequências desconhecidas (e.g. *callbacks*):
  - **obj^msg(args)** – a operação enviou (*has sent*) uma instância da mensagem *msg* (chamada de operação ou envio de sinal) para o objecto *obj*
  - *Wildcard* "?" pode ser usado para argumentos de valor desconhecido ou livre
- Exemplo:
  - context Clock::IncrCurDate()
    post: alarms(curdate)@pre->forAll(alarm | alarm ^ AlarmFired())
  - Mas não garante que é seguida a ordem de registo!
  - E não permite tratar o caso de eliminação/adição de alarmes!

* Exercício

- Escrever uma especificação de um sistema de gestão de elevadores, usando um relógio simulado, com vista a permitir avaliar, por simulação, o desempenho de diferentes algoritmos de atendimento dos pedidos

Concorrência e sincronização

Concorrência e sincronização em VDM++

- Concorrência: através da definição de **objectos activos** que podem ser donos de *threads*
  - Classes de objectos activos têm secção "thread" onde se especifica o comportamento do thread
  - Instrução "start" inicia *thread* num objecto previamente criado
  - Dois tipos de *threads*: simples e periódicos
- Sincronização: através de restrições de sincronização no acesso a **objectos partilhados** (tipicamente passivos)
  - Restrições de sincronização são definidas de forma **declarativa**
  - Permitem limitar concorrência entre objectos activos/threads
  - Restrições são indicadas na secção "sync" da definição da classe
  - Dois tipos de restrições/predicaos: de permissão e de exclusão mútua
  - Restrições são herdadas por subclasses

Threads simples (ou procedimentais)

- thread *statement(s)*
  - Secção da definição da classe que indica a instrução (normalmente uma operação) ou sequência de instruções a realizar pelo *thread*
  - O *thread* morre quando se completa a execução dessa(s) instrução(ões)
- start(*objRef*)
  - Instrução usada para iniciar um *thread* sobre o objecto indicado
  - O *thread* não é iniciado ao criar o objecto para permitir inicializações
  - Chamado de novo (mesmo sem acabar anterior), inicia novo *thread*
- startlist(*objRefSet*)
  - Instrução usada para iniciar um conjunto de *threads*
- threadid
  - Número natural que identifica univocamente o *thread* corrente

Exemplo

```
class Worker
 operations
    public doit() == (
      dcl io : IO := new IO();
      dcl rc : bool;
      for i = 1 to 40 do
          rc := io.fwriteval[nat * nat]("out.txt",
                    mk_(threadid, i), <append>)
    );
    public wait_done() == skip;
    public static main() == (
      dcl w1 : Worker := new Worker();
      dcl w2 : Worker := new Worker();
      start(w1); start(w2);
      w1.wait_done(); w2.wait_done()
    );
 thread doit()
 sync per wait_done => #fin(doit) > #act(wait_done)
end Worker
```

standard VDM++ IO library

Com w1 outra vez também resultava

out.txt

mk_(2, 1)	mk_(3, 9)	mk_(2, 37)
mk_(2, 2)	mk_(3, 10)	mk_(2, 38)
mk_(2, 3)	mk_(3, 11)	mk_(2, 39)
mk_(2, 4)	mk_(3, 12)	mk_(2, 40)
mk_(2, 5)	mk_(3, 13)	mk_(3, 21)
mk_(2, 6)	mk_(3, 14)	mk_(3, 22)
mk_(2, 7)	mk_(3, 15)	mk_(3, 23)
mk_(2, 8)	mk_(3, 16)	mk_(3, 24)
mk_(2, 9)	mk_(3, 17)	mk_(3, 25)
mk_(2, 10)	mk_(3, 18)	mk_(3, 26)
mk_(2, 11)	mk_(3, 19)	mk_(3, 27)
mk_(2, 12)	mk_(3, 20)	mk_(3, 28)
mk_(2, 13)	mk_(2, 21)	mk_(3, 29)
mk_(2, 14)	mk_(2, 22)	mk_(3, 30)
mk_(2, 15)	mk_(2, 23)	mk_(3, 31)
mk_(2, 16)	mk_(2, 24)	mk_(3, 32)
mk_(2, 17)	mk_(2, 25)	mk_(3, 33)
mk_(2, 18)	mk_(2, 26)	mk_(3, 34)
mk_(2, 19)	mk_(2, 27)	mk_(3, 35)
mk_(2, 20)	mk_(2, 28)	mk_(3, 36)
mk_(3, 1)	mk_(2, 29)	mk_(3, 37)
mk_(3, 2)	mk_(2, 30)	mk_(3, 38)
mk_(3, 3)	mk_(2, 31)	mk_(3, 39)
mk_(3, 4)	mk_(2, 32)	mk_(3, 40)
mk_(3, 5)	mk_(2, 33)	
mk_(3, 6)	mk_(2, 34)	
mk_(3, 7)	mk_(2, 35)	
mk_(3, 8)	mk_(2, 36)	

A propósito: biblioteca padrão de IO

- Incluir ficheiro $TOOLBOXHOME/stdlib/io.vpp no projecto
- writeval[tipo](valor)
  - Função que escreve o valor do tipo indicado, em ASCII, no standard output
  - Exemplo: writeval[nat](20)
- fwriteval[tipo](ficheiro, valor, modo)
  - Função que escreve o valor do tipo indicado, em ASCII, no standard output
  - O modo pode ser <append> (acrescentar) ou <start> (criar)
  - Exemplo: fwriteval[nat]("output.txt", 20, <append>)
- echo(texto)
  - Operação que escreve o texto, possivelmente com sequências de escape, no standard output.
  - Exemplo: echo("ola\n")
- fecho(ficheiro, texto, [modo])
  - Idem, em ficheiro
- ferror()
  - Todas as funções/operações anteriores devolvem false em caso de erro. Esta operação devolve (e limpa) a string com a mensagem de erro correspondente

Threads periódicos

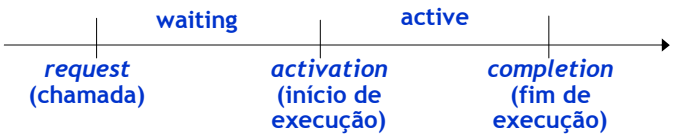
- thread periodic (*timeinterval*) (*opname*)
  - Executa repetidamente a operação de acordo com o intervalo de tempo especificado (em "unidades de tempo do sistema")
  - A operação deve executar em tempo inferior ao intervalo de tempo
  - Não são suportados pelas VDMTools Light

```
-- timer that periodically increments its clock, at every 1000 system time units
class Timer
 instance variables
   private curTime : nat := 0;
 operations
   private IncTime() == curTime := curTime + 1;
   public GetTime() res: nat == return curTime;
 thread
    periodic(1000)(IncTime)
end Elevator
```

Predicados de permissão

- per *operation-name* => *guard-condition*
  - Especifica condição a verificar para a operação poder ser executada
  - Se não se verificar no momento da chamada da operação, esta fica em espera
- Condição de guarda pode usar valores de variáveis de instância, bem como valores de contadores de execução de operações (ver a seguir)
- Condição de guarda é diferente de uma pré-condição
  - Não satisfação de pré-condição é um erro
  - Não satisfação de condição de guarda apenas coloca a chamada em espera
- Interpretador detecta e avisa eventuais situações de "deadlock"
- Só se pode especificar um predicado de permissão por operação
- * Regras para reavaliação das condições de guarda:
  - Ocorre quando termina a execução duma operação (sobre mesmo objecto)
  - Teste da condição e (potencial) activação da operação realizados atomicamente
  - Não está definido qual é o objecto cuja expressão de guarda é reavaliada 1º

Contadores de execução de operações



Expressão	Descrição
#act(*op- name*)	Nº de vezes que a operação foi activada (iniciou execução) sobre este objecto.
#fin(*op- name*)	Nº de vezes que a operação foi concluída (terminou execução) sobre este objecto.
#active(*op- name*)	Nº de chamadas da operação que estão presentemente activas sobre este objecto. #active(op-name) = #act(op-name) - #fin(op-name)
#req(*op- name*)	Nº de chamadas da operação sobre este objecto.
#waiting(*op- name*)	Nº de chamadas que estão presentemente em espera sobre este objecto. #waiting(op-name) = #req(op-name) - #act(op-name)

Predicados de exclusão mútua (mutex)

- mutex(*op-name1*, *op-name2*, …)
  - Operações não podem executar em simultâneo (sobre o mesmo objecto)
- mutex(all)
  - "all" refere-se a todas as operações definidas na classe e superclasses
- A mesma operação pode aparecer em múltiplos predicados mutex (e num predicado de permissão)
- Predicados mutex são implicitam/ traduzidos para predicados de permissão

```
mutex(opA, opB);
mutex(opB, opC, opD);
per opD => someVariable > 42;
```

```
per opA => #active(opA) + #active(opB) = 0;
per opB => #active(opA) + #active(opB) = 0 and
           #active(opB) + #active(opC) + #active(opD) = 0;
per opC => #active(opB) + #active(opC) + #active(opD) = 0;
per opD => #active(opB) + #active(opC) + #active(opD) = 0
           and someVariable > 42;
```


Exemplo – Bounded Buffer (1/4)

```
-- Bounded buffer used to exchange messages between active objects
class BoundedBuffer
values
   public EndOfMessage : char = '!';
instance variables
   private buf : seq of char := [];
   private size : nat1;
operations
  public BoundedBuffer(sz: nat1) res:BoundedBuffer == (size := sz; return self);
  public Get() res : char == (dcl c : char := hd buf;   buf := tl buf; return c );
  public Put(c: char) == ( buf := buf ^ [c] );
sync
   per Get => len buf > 0;     -- waits until buffer not empty
   per Put => len buf < size; -- waits until buffer not full
end BoundedBuffer
```

Sistema sequencial → Sistema concorrente
Pré-condição (c/args) → Condição de guarda (s/args)

Exemplo - Bounded Buffer (2/4)

```
-- Active object responsible for sending a message through a channel
class SenderAgent
 instance variables
   private channel : BoundedBuffer;
   private message : seq of char;
 operations
   public SenderAgent(ch: BoundedBuffer, msg: seq of char)res:SenderAgent ==
     (channel := ch; message := msg; return self);
   public SendNow() == (
     for c in message do channel.Put(c);
     channel.Put(BoundedBuffer`EndOfMessage)
    )
 thread SendNow()
end SenderAgent
```


Exemplo - Bounded Buffer (3/4)

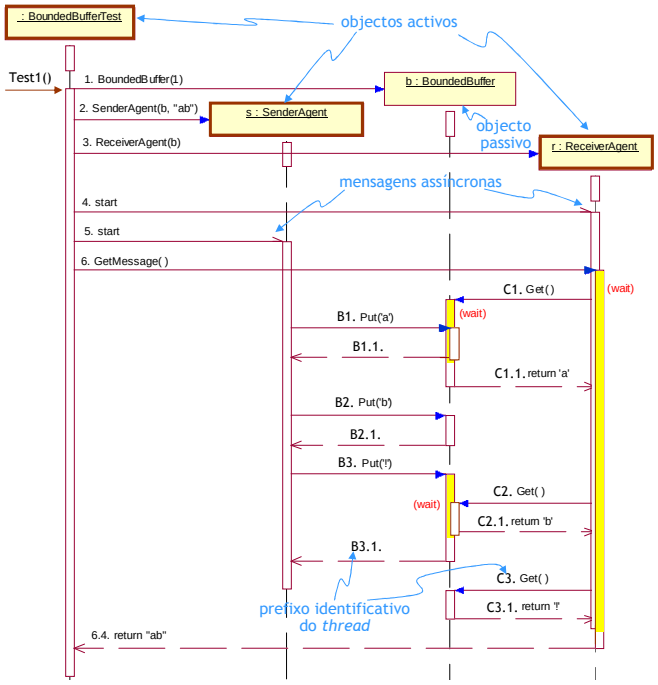
```
-- Active object responsible for receiving a message through a channel
class ReceiverAgent
  instance variables
      private channel : BoundedBuffer;
      private message : seq of char := [];
  operations
       public ReceiverAgent(ch : BoundedBuffer) r: ReceiverAgent ==
         (channel := ch; return self);
       public ReceiveNow() == (
           dcl c : char := channel.Get();
           message := [];
           while c <> BoundedBuffer`EndOfMessage do
               ( message := message ^ [c];  c := channel.Get())
         );
       public GetMessage() res : seq of char == return message;
  thread ReceiveNow()
  sync -- GetMessage waits for the reception of one more message
        per GetMessage =>  #fin(ReceiveNow) > #act(GetMessage)
end ReceiverAgent
```

Exemplo - Bounded Buffer (4/4)

```
class BoundedBufferTest is subclass of Test
  operations
   -- método de teste que comando o envio e recepção duma mensagem
    public Test1() == (
          dcl m : seq of char := "ab";
          dcl b : BoundedBuffer := new BoundedBuffer(1);  -- capac<len msg!
          dcl s : SenderAgent := new SenderAgent(b, m);
          dcl r : ReceiverAgent := new ReceiverAgent(b);
          start(r);
          start(s);
          Assert(r.GetMessage() = m)
    )
end BoundedBufferTest
```


Exemplo de execução

(diagrama de sequência UML modificado)



* Exercícios

- Experimentar os exemplos descritos nas VDM Tools
- Especificar um gestor de *locks* partilhados (para leitura) e exclusivos (para escrita) a objectos, com ou sem espera

Projecto: Máquina de vendas

- Elaborar um modelo UML (diagramas de classes e de estados) de uma máquina de vendas, importar para VDM++, e completar especificação (para já não executável) com invariantes, pré-condições, pós-condições e corpo de operações e testar a especificação.
- Requisitos
  - A máquina aceita moedas de 0.05, 0.1, 0.2, 0.5, 1 e 2 euros
  - A máquina tem um stock de moedas que servem para dar o troco aos clientes que o necessitem
  - A máquina tem um stock de produtos e cada produto tem um preço
  - A máquina pode estar em modo de operação ou manutenção
  - No modo de manutenção, é possível actualizar o stock de produtos e de moedas
  - No modo de utilização, a máquina deve mostrar aos clientes os produtos disponíveis; estes seleccionam o produto pretendido e depois inserem as moedas correspondentes; a máquina deve dar troco sempre que possível ou devolver o dinheiro ao cliente sem efectuar a venda.
- Usar algoritmo guloso (*greedy*) com retrocesso (*backtracking*) para calcular o troco procurando minimizar o nº de moedas

Referências e leituras adicionais

- Validated Designs for Object-oriented Systems. John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat and Marcel Verhoef. ISBN: 1-85233-881-4. Springer Verlag, New York. 2005.
- http://www.vdmbook.com/
- Manual de VDM++ (langmanpp_a4.pdf)
- Manual de VDMTools (usermanpp_a4.pdf)
- www.uml.org – especificações e recursos sobre UML, OCL, etc.

Anexo A: Guião de Utilização da Ferramenta VDMTools Lite

Utilização das VDMTools Lite

- Instalar
- Arrancar
- Criar um projecto
- Escrever uma especificação
- Adicionar ficheiro ao projecto
- Verificar sintaxe e tipos
- Depuração de erros
- Correr o interpretador
- Verificação de invariantes, pré-condições e pós-condições
- Reunir casos de teste numa classe de teste
- Analisar cobertura dos testes
- Produzir um relatório do projecto

Instalar

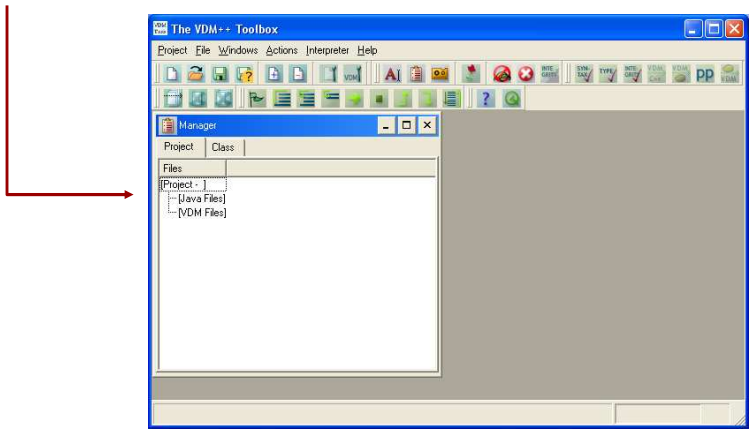
- VDM++ Toolbox Lite v8.0 (versão sem geração de código)
- Download gratuito (após registo) a partir de http://www.vdmbook.com/ -> tools -> download
- Instalação em Linux: a versão Lite (gratuita) só existe para Windows, a versão comercial também existe para Linux

Arrancar

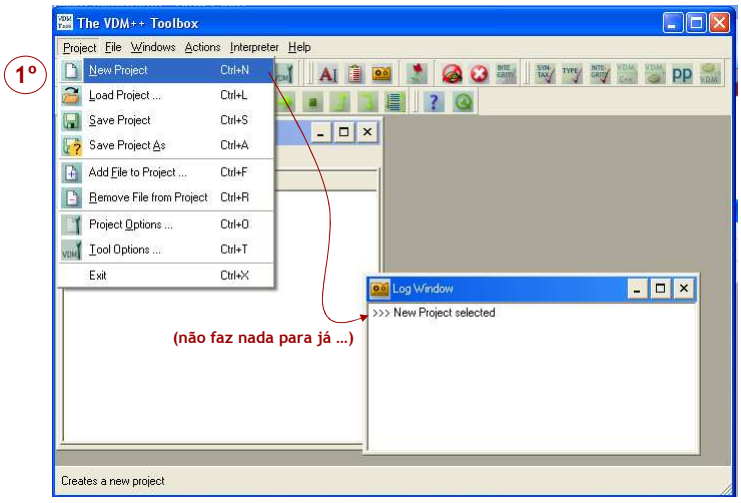
Start -> VDMTools -> VDM++ Toolbox Lite v8.0

ou

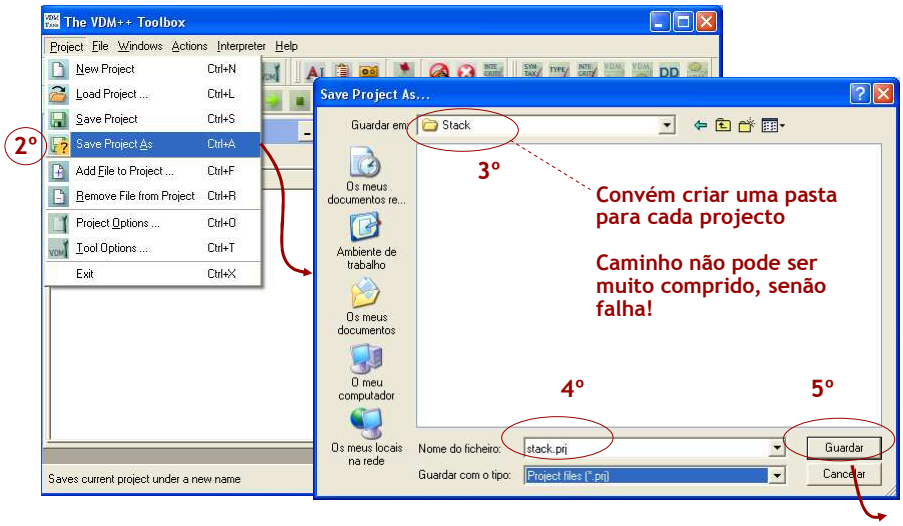
The VDM++ Toolbox Lite v8.0\bin\vppgde.exe



Criar um projecto (1/2)

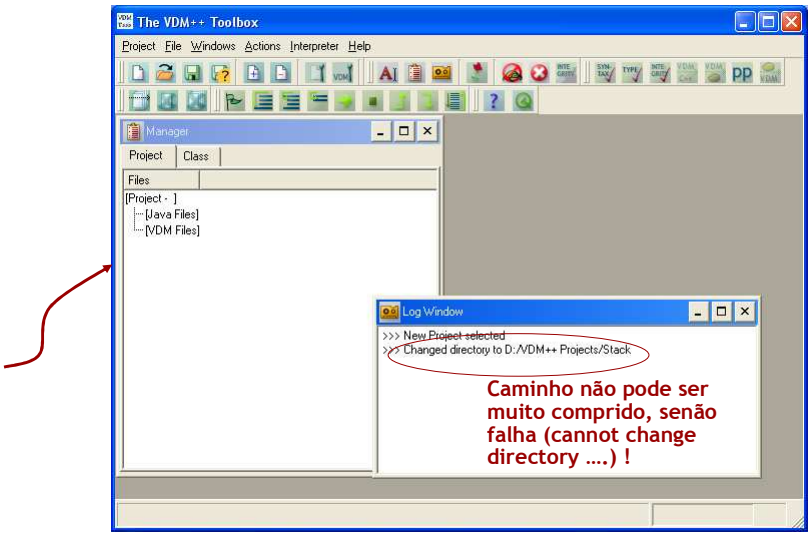


Criar um projecto (2/2)



(versão de Windows em português ...)

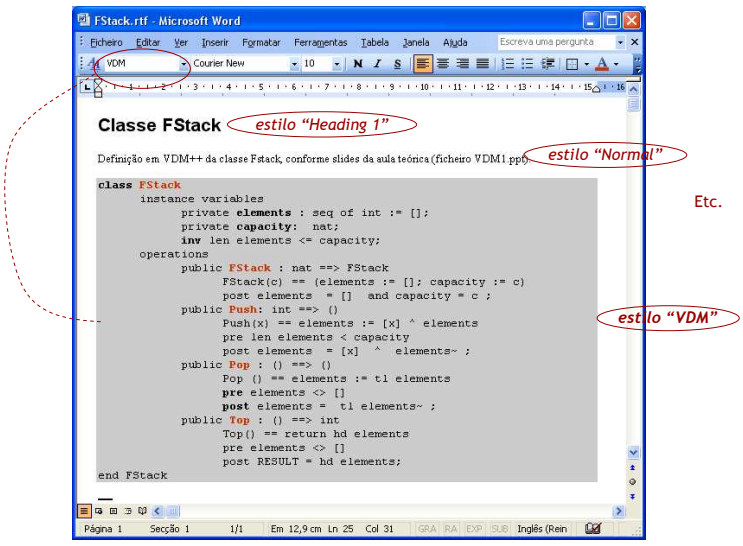
Criar um projecto (3/3)



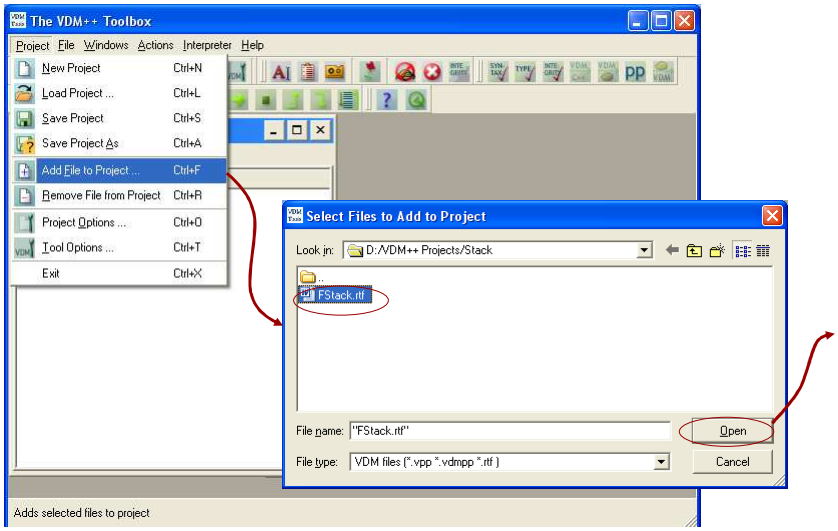
Escrever uma especificação (1)

- Programação literária (código e documentação integrados)
- Código em VDM++ é escrito num documento em formato RTF (Rich Text Format), usando estilos pré-definidos no template "The VDM++ Toolbox Lite v8.0\word\VDM.dot"
- Abrir o ficheiro "VDM.dot" e gravar em formato RTF, com o nome da classe a criar, por exemplo "FStack.rtf", no directório do projecto
- Normalmente, convém criar um ficheiro para cada classe
- O ficheiro pode ter código VDM++ e documentação
- Formatar o código VDM++ com o estilo "VDM"

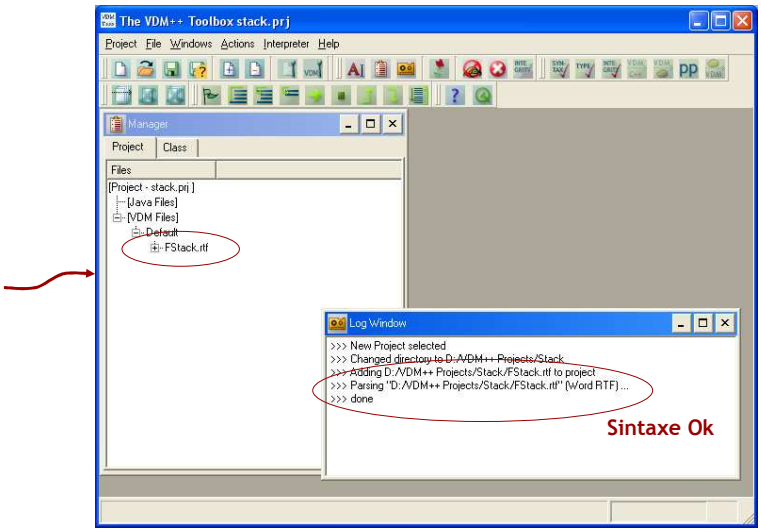
Escrever uma especificação (2)



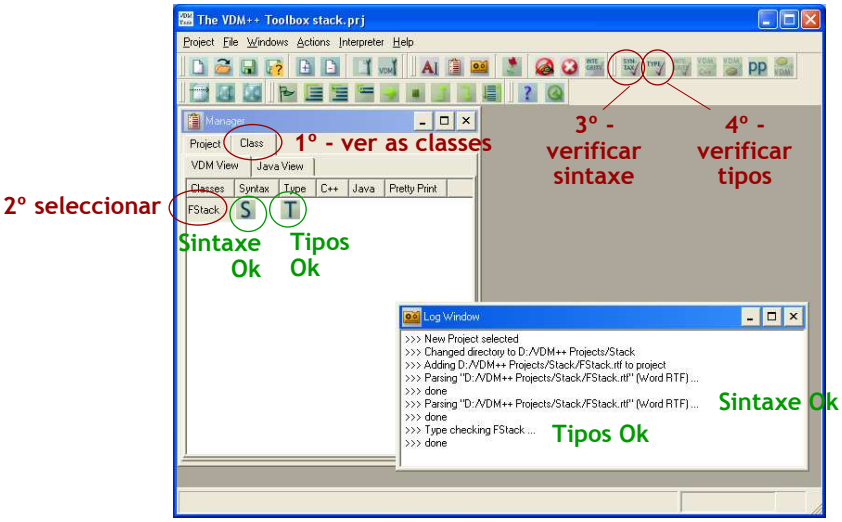
Adicionar ficheiro ao projecto (1)



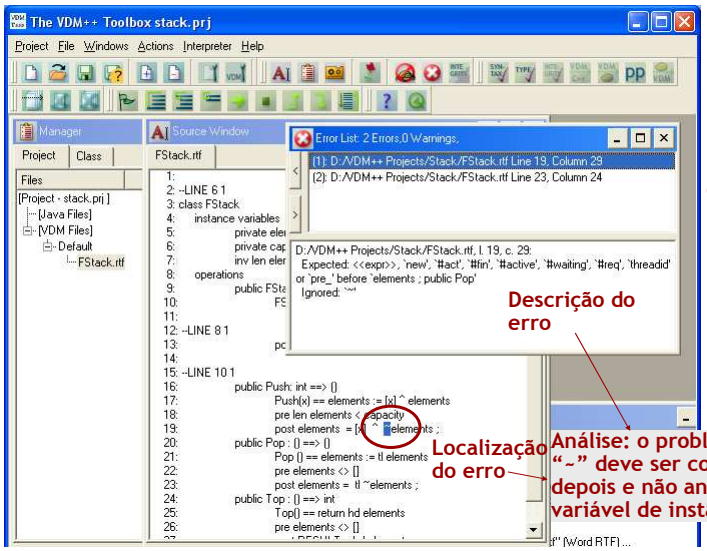
Adicionar ficheiro ao projecto (2)



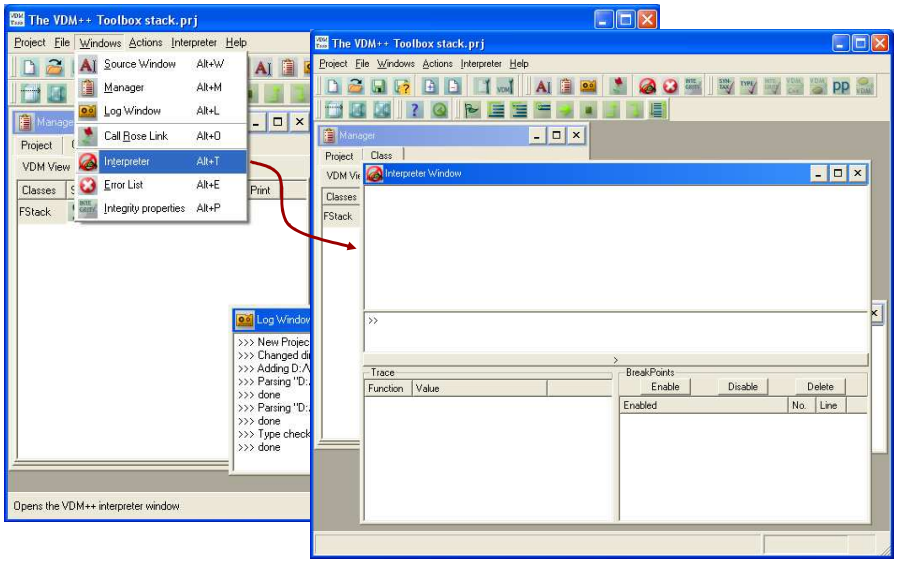
Verificar sintaxe e tipos



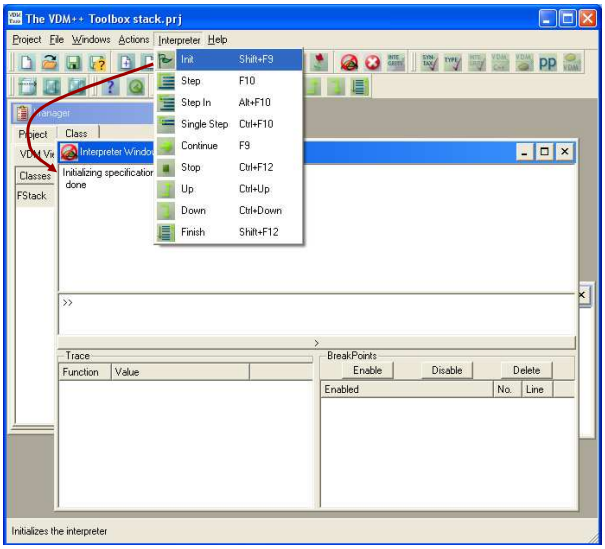
Depuração de erros



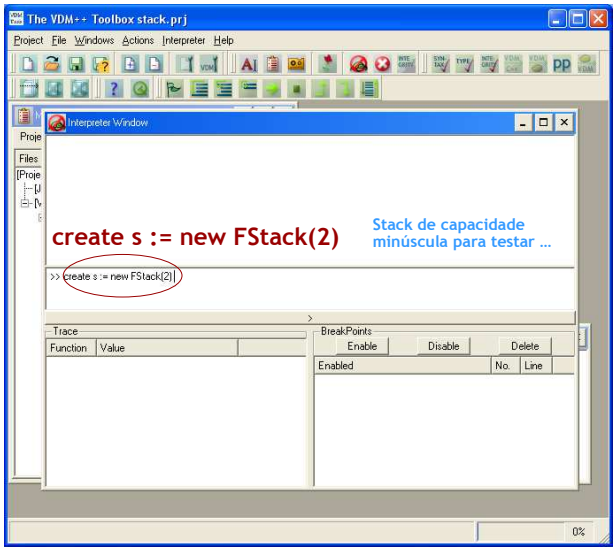
Correr o interpretador - abrir



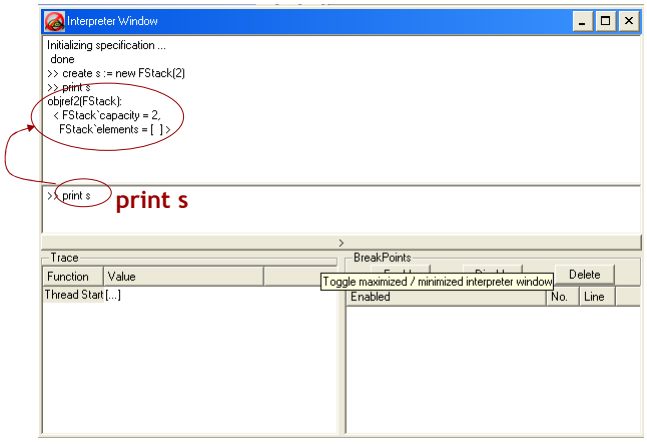
Correr o interpretador - inicializar



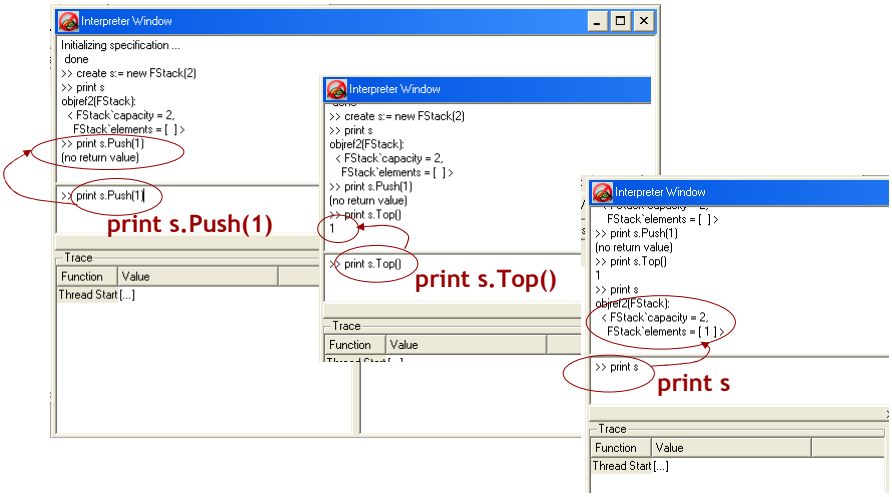
Correr o interpretador - criar um objecto



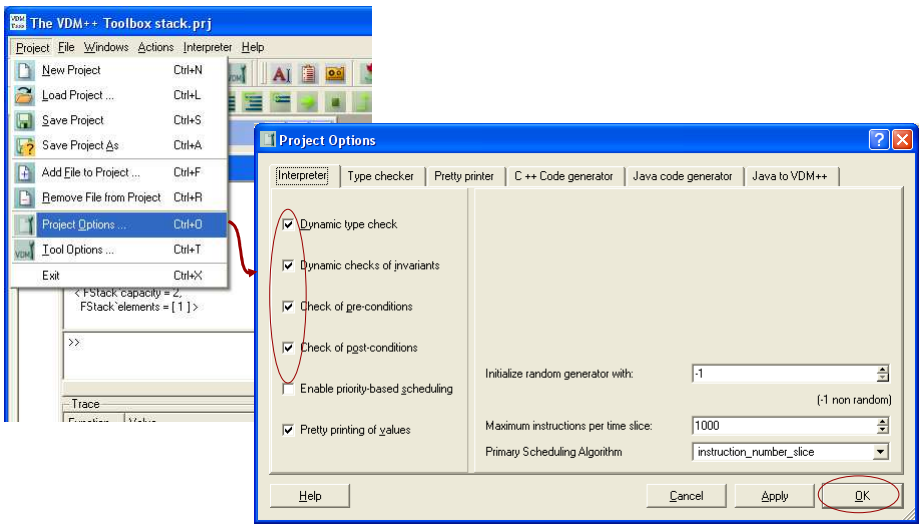
Correr o interpretador - consultar o estado dum objecto



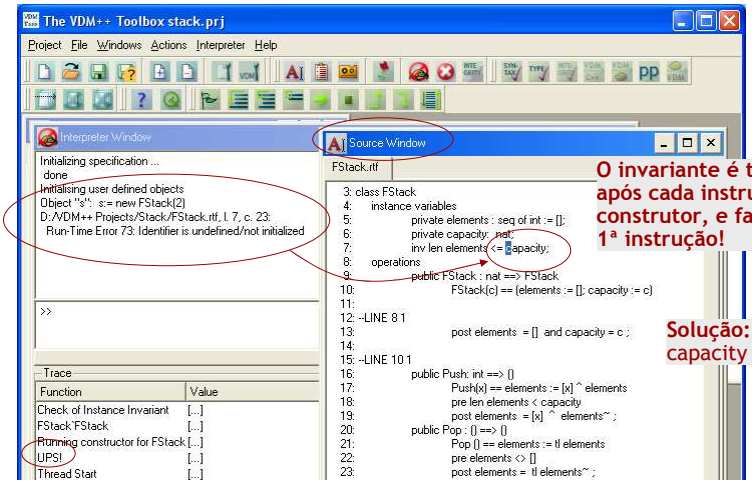
Correr o interpretador - manipular um objecto



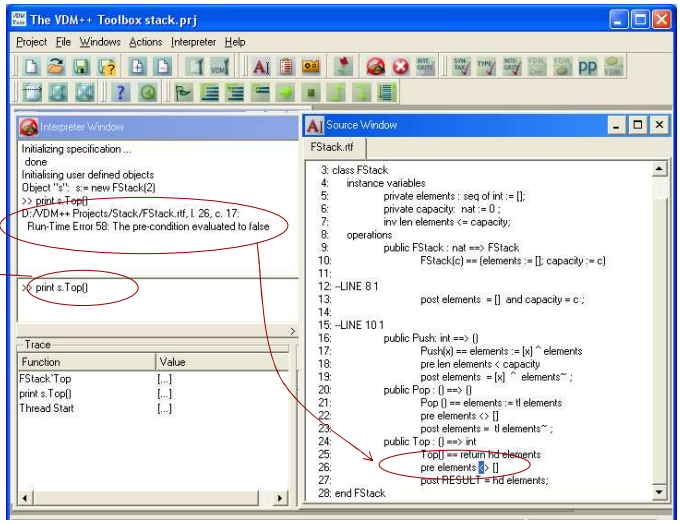
Verificação de invariantes, pré-condições e pós-condições - Activar verificação



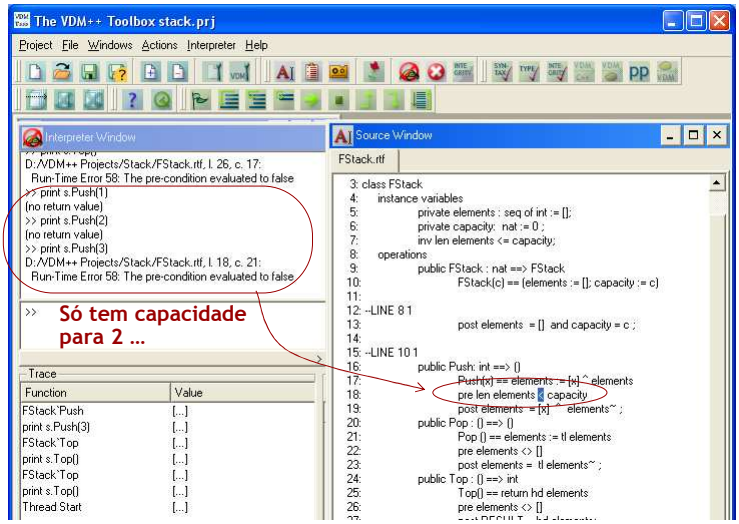
Verificação de invariantes, pré-condições e pós-condições - Violação de invariante



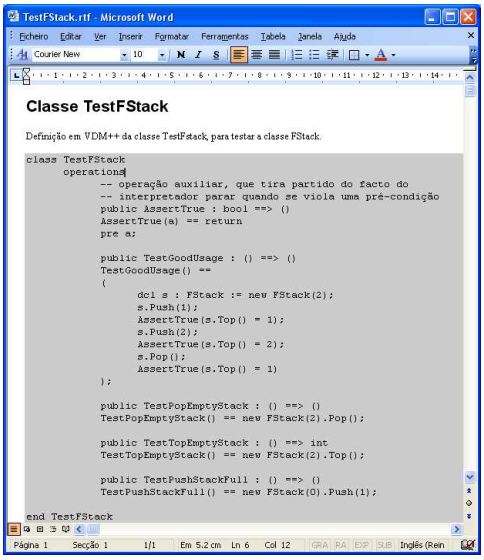
Verificação de invariantes, pré-condições e pós-condições - Violação de pré-condição



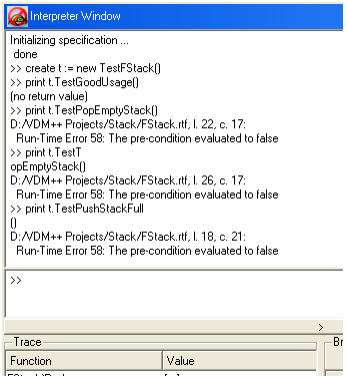
Verificação de invariantes, pré-condições e pós-condições - Violação de pré-condição



Reunir casos de teste numa classe de teste

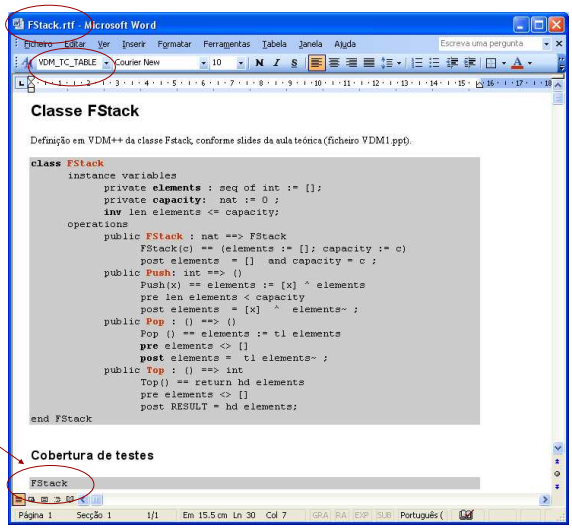


1. Criar o ficheiro
2. Adicionar ao projecto
3. Verificar sintaxe
4. Verificar tipos
5. Inicializar o interpretador
6. Executar os seguintes comandos:



Analisar cobertura dos testes

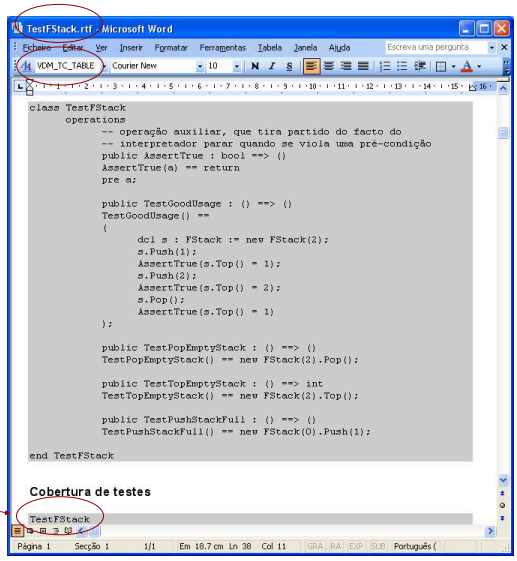
- 1º - Colocar *placeholders* para tabelas de cobertura de testes (*test coverage*)



Nome da classe, estilo VDM_TC_TABLE.

Será substituído por tabela com informação de cobertura de testes ao fazer "pretty print".

Analisar cobertura dos testes



Pode-se fazer o mesmo relativamente ao próprio código de teste …

Analisar cobertura dos testes

- 2º - Para cada caso de teste XPTO, preparar dois ficheiros de texto (por exemplo no directório do projecto):
  - ficheiro XPTO.arg, com o comando para o interpretador
  - ficheiro XPTO.arg.exp, com o resultado esperado da avaliação do comando (print comando)

Input **Output esperado**

Etc.

Na realidade, deve violar pré-condição, mas não há maneira de o indicar (!?)

Analisar cobertura dos testes

- 3º - Preparar *scripts* de teste (no mesmo directório)

vdmloop.bat - Bloco de notas

```
@echo off
rem Runs a collection of VDM++ test examples
rem Assumes specification is in Word RTF files

set S1=FStack.rtf
set S2=TestFStack.rtf

"D:\Programas\The VDM++ Toolbox v6.8.6-Lite\bin\vppde" -p -R vdm.tc %S1% %S2%
for /R %%f in (*.arg) do call vdmtest "%%f"
```

vdmtest.bat - Bloco de notas

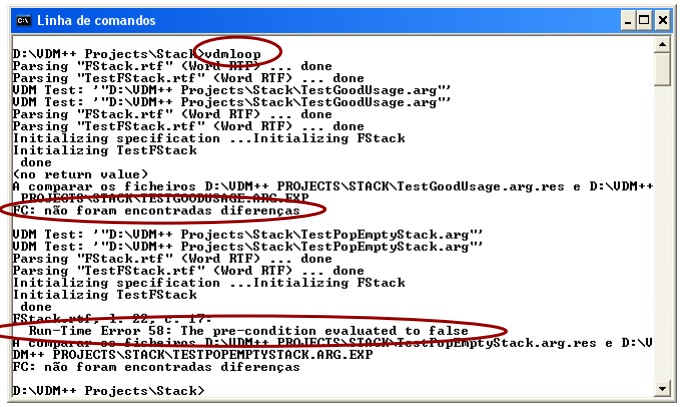
```
@echo off
rem Runs a single teste case
rem -- Output the argument to stdout (for redirect) and "con" (for user feedback)
echo VDM Test: '%1' > con
echo VDM Test: '%1'

rem short names for specification files in Word RTF Format
set S1=FStack.rtf
set S2=TestFStack.rtf

rem -- Calls the interpreter for this test case
"D:\Programas\The VDM++ Toolbox v6.8.6-Lite\bin\vppde" -i -D -I -P -Q -R vdm.tc -O %1.res %1 %S1% %S2%

rem -- Check for difference between result of execution and expected result.
if EXIST %1.exp fc /w %1.res %1.exp

:end
```

Analisar cobertura dos testes

- 4º - Executar o script com o ciclo de testes

OK

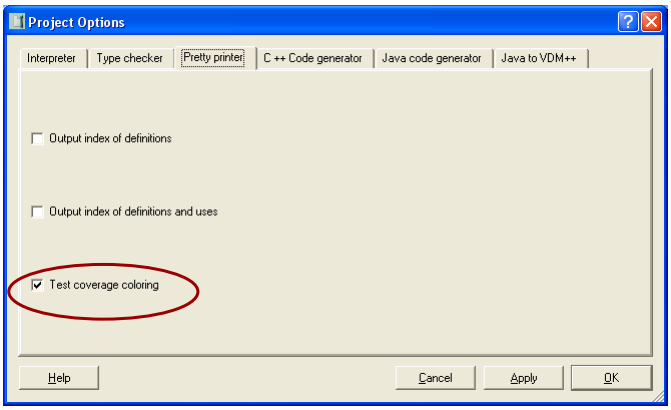
Conforme esperado



Também gera um ficheiro "vdm.tc" com informação de cobertura de código, a usar pelo "pretty printer"!

Analisar cobertura dos testes

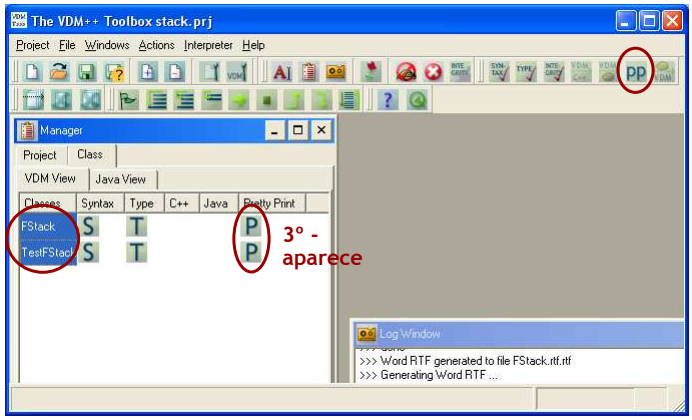
- 5º - Preparar configuração de "pretty printer"



Analisar cobertura dos testes

- 6º - Efectuar o "pretty print"

2º - pretty print

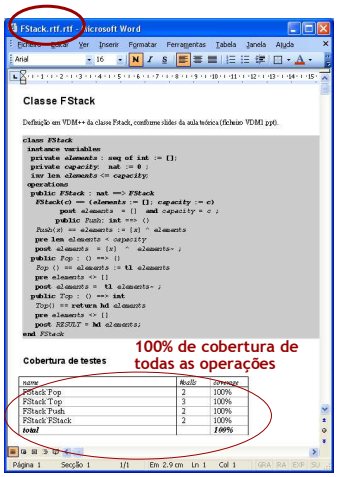


1º - seleccionar

3º - aparece

Analisar cobertura dos testes

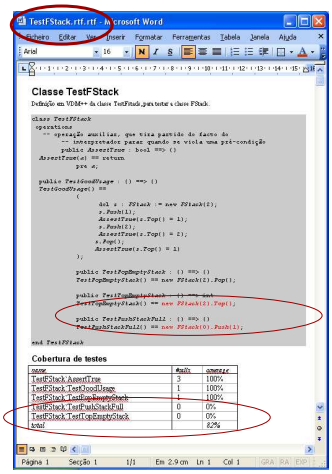
- 7º - Analisar o resultado (ficheiros .rtf.rtf !)



100% de cobertura de todas as operações

Código não executado assinalado a vermelho

duas operações (casos de teste) não foram executadas



Produzir um relatório do projecto

- No Word, criar documento, por exemplo "master.rtf", em que se inserem links para os vários ficheiros com a facilidade "Insert File …" -> botão "Insert as Link"
- Convém inserir os ficheiros resultantes de pretty print (*ClassName*.rtf.rtf)
- Usar facilidade "Update field" para actualizar os links
- Ver exemplo em "Stack\master.rtf"
- Ver exemplo com todos os ficheiros do projecto em Stack.zip

Anexo B: Operadores e expressões em VDM++

Operadores e expressões

- Operadores
  - Operadores de comparação
  - Operadores booleanos
  - Operadores numéricos
  - Operadores sobre conjuntos
  - Construção de conjuntos
  - Operadores sobre sequências
  - Construção de sequências
  - Operadores sobre mapeamentos
  - Construção de mapeamentos
  - Operadores sobre *records*
  - Operadores sobre tuplos
- Expressões
  - Expressão “let”
  - Expressões “if” e “cases”
  - Padrões
  - Quantificadores
  - Ligações *(bindings)*
  - Expressão de selecção “iota”
  - Teste de pertença a tipo
  - Teste de pertence a classe

Operadores de comparação

- Igual: `=`
- Diferente: `<>`
- Estão definidos para todos os tipos de dados, inclusive conjuntos, sequências e mapeamentos (*maps*)
- São os únicos operadores para os tipos char, token e *quotes*

Operadores booleanos

Operator	Name	Type
not b	Negation	bool → bool
a and b	Conjunction	bool * bool → bool
a or b	Disjunction	bool * bool → bool
a => b	Implication	bool * bool → bool
a <=> b	Biimplication	bool * bool → bool
a = b	Equality	bool * bool → bool
a <> b	Inequality	bool * bool → bool

(fonte: langmanpp_a4.pdf)

- No caso de "and", "or" e "=>", 2º operando só é avaliado se for necessário para determinar o resultado
- Na realidade, segue lógica de 3 valores (3º valor significa não definido)
  - ver detalhes no manual

Operadores numéricos

Nota:
Os tipos indicados para os operandos são os mais gerais permitidos.
Isto significa, por exemplo, que o sinal menos unário funciona para operandos de todos os tipos numéricos (nat1, nat, int, rat e real).

(fonte: langmanpp_a4.pdf)

Operator	Name	Type
-x	Unary minus	real → real
abs x	Absolute value	real → real
floor x	Floor	real → int
x + y	Sum	real * real → real
x - y	Difference	real * real → real
x * y	Product	real * real → real
x / y	Division	real * real → real
x div y	Integer division	int * int → int
x rem y	Remainder	int * int → int
x mod y	Modulus	int * int → int
x**y	Power	real * real → real
x < y	Less than	real * real → bool
x > y	Greater than	real * real → bool
x <= y	Less or equal	real * real → bool
x >= y	Greater or equal	real * real → bool
x = y	Equal	real * real → bool
x <> y	Not equal	real * real → bool

Operadores sobre conjuntos

Operador	Nome	Descrição	Tipo
e in set s1	Pertença	$e \in s1$	A * set of $A \to$ bool
e not in set s1	Não pertença	$e \notin s1$	
s1 union s2	Reunião	$s1 \cup s2$	set of A * set of $A \to$ set of A
s1 inter s2	Intersecção	$s1 \cap s2$	
s1 \ s2	Diferença	$s1 \setminus s2$	
s1 subset s2	Subconjunto	$s1 \subseteq s2$	set of A * set of $A \to$ bool
s1 psubset s2	Subconjunto próprio	$s1 \subset s2$ $(s1 \subseteq s2 \wedge s1 \neq s2)$	
s1 = s2	Igualdade	$s1 = s2$	
s1 <> s2	Desigualdade	$s1 \neq s2$	
card s1	Cardinal	# s1	set of $A \to$ nat
dunion ss	Reunião distribuída	$\bigcup_{s_i \in ss} s_i$	set of set of $A \to$ set of A
dinter ss	Intersecção distribuída	$\bigcap_{s_i \in ss} s_i$	
power s1	Conjunto de sub-conjuntos (ou partes) de s1	$\mathscr{P}(s1)$	set of $A \to$ set of set of A

Operadores sobre conjuntos

Examples: Let s1 = {<France>,<Denmark>,<SouthAfrica>,<SaudiArabia>}, s2 = {2, 4, 6, 8, 11} and s3 = {} then:

```
<England> in set s1                          ≡ false
10 not in set s2                             ≡ true
s2 union s3                                  ≡ {2, 4, 6, 8, 11}
s1 inter s3                                  ≡ {}
(s2 \ {2,4,8,10}) union {2,4,8,10} = s2      ≡ false
s1 subset s3                                 ≡ false
s3 subset s1                                 ≡ true
s2 psubset s2                                ≡ false
s2 <> s2 union {2, 4}                        ≡ false
card s2 union {2, 4}                         ≡ 5
dunion {s2, {2,4}, {4,5,6}, {0,12}}          ≡ {0,2,4,5,6,8,11,12}
dinter {s2, {2,4}, {4,5,6}}                  ≡ {4}
dunion power {2,4}                           ≡ {2,4}
dinter power {2,4}                           ≡ {}
```

Construção de conjuntos

- Em extensão: *{e1, e2, …, en}*
  - Exemplos: {1, 3, 5}, { }
- Em compreensão: *{e | bd1, bd2, ..., bdm & P}*
  - Conjunto formado pelos valores da expressão *e* (sem duplicados) para todos os valores das variáveis introduzidas nos *bindings* que obedecem ao predicado *P*
  - Normalmente o *binding* é da forma: *variável* in set *conjunto*
  - Formas mais gerais de bindings – ver mais adiante
  - A parte do predicado (& *P*) é opcional
- Por intervalo: *{lower, …, upper}*
  - Conjunto de valores inteiros entre *lower* e *upper*, inclusive
  - Exemplo: {1, ... , 6} é o mesmo que {1, 2, 3, 4, 5, 6}
  - Se *lower* > *upper* , o conjunto é vazio

Exemplo: Registo Civil

- Operação para obter os filhos de uma pessoa:

```
public GetFIlhos() res : set of Pessoa ==
  return { f | f in set pessoas & f.pai = self or f.mae = self };
```

- Operação p/ obter todos os descendentes de uma pessoa:

```
public GetDescendentes() res : set of Pessoa ==
 return GetFilhos() union
          dunion { f.GetDescendentes() | f in set GetFilhos() };
```

Operadores sobre sequências

Operador	Nome	Descrição	Tipo
hd l	Cabeça (*head*)	Dá o 1º elemento de l, que não pode ser vazia	seq of *A* → *A*
tl l	Cauda (*tail*)	Dá a subsequência de l em que o 1º elemento foi removido. l não pode ser vazia	seq of *A* → seq of *A*
len l	Comprimento	Dá o comprimento de l	seq of *A* → nat
elems l	Elementos	Dá o conjunto formado pelos elementos de l (sem ordem nem repetidos)	seq of *A* → set of *A*
inds l	Índices	Dá o conjunto dos índices de l, i.e., {1, ..., len l}	seq of *A* → set of nat1
l1 ^ l2	Concatenação	Dá a sequência formada pelos elementos de l1 seguida pelos elementos de l2	(seq of *A*) * (seq of *A*) → seq of *A*
conc ll	Concatenação distribuída	Dá a sequência formada pela concatenação dos elementos de ll (que são por sua vez sequências)	seq of seq of *A* → seq of *A*
l ++ m	Modificação de sequência	Os elementos de l cujos índices estão no domínio de m são modificados para o valor correspondente em m. Deve-se verificar: dom m subset inds l.	(seq of *A*) * (map nat1 to *A*) → seq of *A*
l(i)	Aplicação de sequência	Dá o elemento que se encontra no índice i de l. Deve-se verificar: i in set inds l.	seq of *A* * nat1 → *A*
l(i, ..., j)	Subsequência	Dá a subsequência de l entre os índices i e j, inclusive. Se i < 1, considera-se 1. Se j > len s, considera-se len(s).	seq of *A* * nat * nat → seq of *A*

Outros: =, <>

Operadores sobre sequências

Examples: Let l1 = [3,1,4,1,5,9,2], l2 = [2,7,1,8],
l3 = [<England>, <Rumania>, <Colombia>, <Tunisia>] then:

```
len l1                                  ≡  7
hd (l1^l2)                              ≡  3
tl (l1^l2)                              ≡  [1,4,1,5,9,2,2,7,1,8]
l3(len l3)                              ≡  <Tunisia>
"England"(2)                            ≡  'n'
conc [l1,l2] = l1^l2                    ≡  true
conc [l1,l1,l2] = l1^l2                 ≡  false
elems l3                                ≡  { <England>, <Rumania>,
                                             <Colombia>,<Tunisia>}
(elems l1) inter (elems l2)             ≡  {1,2}
inds l1                                 ≡  {1,2,3,4,5,6,7}
(inds l1) inter (inds l2)               ≡  {1,2,3,4}
l3 ++ {2 |-> <Germany>,4 |-> <Nigeria>} ≡  [ <England>, <Germany>,
                                             <Colombia>, <Nigeria>]
l1(2, ..., 4)                           ≡  [1, 4, 1]
```

Construção de sequências

- Em extensão: [e1, e2, …, en]
  - Exemplos: [1, 3, 1, 5], [ ]
- Em compreensão: [*e* | *id* in set *S* & *P*]
  - Constrói a sequência formada pelos valores da expressão *e* para todos os valores do identificador *id* (por ordem crescente) em que o predicate *P* é verdadeiro.
  - A expressão *e* deve usar o identificador *id*
  - *S* deve ser um conjunto numérico
  - A parte do predicado (& *P*) é opcional

Operadores sobre mapeamentos

Operador	Nome	Descrição	Tipo
dom m	Domínio	Dá o domínio (conjunto de chaves) de m	map *A* to *B* → set of *A*
rng m	Contra-domínio (*range*)	Dá o contra-domínio (conjunto de valores correspondentes a chaves) de m	map *A* to *B* → set of *B*
m1 munion m2	Reunião (*merge*)	Faz a reunião dos pares *chave-valor* existentes em m1 e m2, que têm de ser compatíveis (não podem fazer corresponder valores diferentes a chaves iguais)	(map *A* to *B*) * (map *A* to *B*) → map *A* to *B*
m1 ++ m2	Sobreposição (*override*)	Reunião sem restrição de compatibilidade. Em caso de conflito, prevalece m2.	
merge ms	Reunião distribuída	Faz a reunião dos mapeamentos contidos em ms, que devem ser compatíveis.	set of (map *A* to *B*) → map *A* to *B*

Operadores sobre mapeamentos

Operador	Nome	Descrição	Tipo
s <: m	Domínio restrito a	Dá o mapeamento constituído pelos elementos de `m` cuja chave está em `s` (que não tem de ser um subconjunto de `dom m`)	(set of A) * (map *A* to *B*) → map *A* to *B*
s <-: m	Domínio restrito por	Dá o mapeamento constituído pelos elementos de `m` cuja chave não está em `s` (que não tem de ser um subconjunto de `dom m`)	
m :> s	Contra-domínio restrito a	Dá o mapeamento constituído pelos elementos de `m` cujo valor de informação está em `s` (que não tem de ser um subconjunto de `rng m`)	(map *A* to *B*) * (set of *B*) → map *A* to *B*
m :-> s	Contra-domínio restrito por	Dá o mapeamento constituído pelos elementos de `m` cujo valor de informação não está em `s` (que não tem de ser um subconjunto de `rng m`)	

Operadores sobre mapeamentos

Operador	Nome	Descrição	Tipo
m(d)	Aplicação de mapeamento	Dá o valor corresponde à chave `d` por `m`. A chave `d` deve existir no domínio de `m`.	(map *A* to *B*) * *A* → *B*
m1 comp m2	Composição de mapeamentos	Dá `m2` seguido de `m1`. O mapeamento resultante tem o mesmo domínio que `m2`. O valor correspondente a cada chave é obtido aplicando primeiro `m2` e depois `m1`. Restrição: `rng m2 subset dom m1`.	(map *B* to *C*) * (map *A* to *B*) → map *A* to *C*
m ** n	Iteração	Composição de `m` consigo próprio `n` vezes. Se `n=0`, dá a função identidade, em que cada elemento do domínio é mapeado para si próprio. Se `n=1`, dá `m`. Se `n>1`, `rng m` deve ser um subconjunto de `dom m`.	(map *A* to *A*) * nat → map *A* to *A*
inverse m	Mapeamento inverso	Dá o inverso de `m`, que deve ser injectivo.	inmap *A* to *B* → inmap *B* to *A*

Outros: =, <>

Operadores sobre mapeamentos

Examples: Let

```
m1 = { <France> |-> 9, <Denmark> |-> 4,
       <SouthAfrica> |-> 2, <SaudiArabia> |-> 1},
m2 = { 1 |-> 2, 2 |-> 3, 3 |-> 4, 4 |-> 1 },
Europe = { <France>, <England>, <Denmark>, <Spain> }
```

then:

`dom m1`	≡	`{<France>, <Denmark>, <SouthAfrica>, <SaudiArabia>}`
`rng m1`	≡	`{1,2,4,9}`
`m1 munion {<England> |-> 3}`	≡	`{<France> |-> 9, <Denmark> |-> 4, <England> |-> 3, <SaudiArabia> |-> 1, <SouthAfrica> |-> 2}`

Operadores sobre mapeamentos

`m1 ++ {<France> |-> 8, <England> |-> 4}`	≡	`{<France> |-> 8, <Denmark> |-> 4, <SouthAfrica> |-> 2, <SaudiArabia> |-> 1, <England> |-> 4}`
`merge{ {<France> |-> 9, <Spain> |-> 4} {<France> |-> 9, <England> |-> 3, <UnitedStates> |-> 1}}`	≡	`{<France> |-> 9, <England> |-> 3, <Spain> |-> 4, <UnitedStates> |-> 1}`
`Europe <: m1`	≡	`{<France> |-> 9, <Denmark> |-> 4}`
`Europe <-: m1`	≡	`{<SouthAfrica> |-> 2, <SaudiArabia> |-> 1}`

Operadores sobre mapeamentos

```
m1 :> {2,...,10}                    ≡  {<France> |-> 9,
                                       <Denmark> |-> 4,
                                       <SouthAfrica> |-> 2}

m1 :-> {2,...,10}                   ≡  {<SaudiArabia> |-> 1}

m1 comp ({"France" |-> <France>})   ≡  {"France" |-> 9}

m2 ** 3                             ≡  {1 |-> 4, 2 |-> 1,
                                        3 |-> 2, 4 |-> 3 }

inverse m2                          ≡  {2 |-> 1, 3 |-> 2,
                                        4 |-> 3, 1 |-> 4 }

m2 comp (inverse m2)                ≡  {1 |-> 1, 2 |-> 2,
                                        3 |-> 3, 4 |-> 4 }
```


Construção de mapeamentos

- Em extensão: {d1 |-> r1, d2 |-> r2, ..., dn |-> rn}
  - Exemplo: { 1 |-> "um", 2 |-> "dois" }
  - Mapeamento vazio: {|->} (para distinguir de conjunto vazio)
- Em compreensão: {ed |-> er | bd1, ..., bdn & P}
  - Constrói um mapeamento avaliando as expressões *ed* (chave) e *er* (valor correspondente à chave) para todos os *bindings* possíveis para os quais o predicado *P* é verdadeiro
  - Exemplo: listagem de pessoas e nº de filhos correspondente, restrita às pessoas com filhos

    { p |-> card p.GetFilhos() | p in set pessoas & p.GetFilhos() <> {} }

Operadores sobre *records*

Constructors: The record constructor: `mk_A(a, b)` where `a` belongs to the type `A1` and `b` belongs to the type `A2`.

Operators:

Operator	Name	Type
`r.i`	Field select	$A * Id \rightarrow Ai$
`r1 = r2`	Equality	$A * A \rightarrow$ bool
`r1 <> r2`	Inequality	$A * A \rightarrow$ bool
`is_A(r1)`	Is	$Id * MasterA \rightarrow$ bool

Semantics of Operators:

Operator Name	Semantics Description
Field select	yields the value of the field with fieldname `i` in the record value `r`. `r` must have a field with name `i`.

Operadores sobre *records*

Examples: Let `Score` be defined as

```
Score :: team : Team
         won : nat
         drawn : nat
         lost : nat
         points : nat;
Team = <Brazil> | <France> | ...
```

and let

```
sc1 = mk_Score (<France>, 3, 0, 0, 9),
sc2 = mk_Score (<Denmark>, 1, 1, 1, 4),
sc3 = mk_Score (<SouthAfrica>, 0, 2, 1, 2) and
sc4 = mk_Score (<SaudiArabia>, 0, 1, 2, 1).
```

Then

```
sc1.team                ≡  <France>
sc4.points              ≡  1
sc2.points > sc3.points ≡  true
is_Score(sc4)           ≡  true
is_bool(sc3)            ≡  false
is_int(sc1.won)         ≡  true
sc4 = sc1               ≡  false
sc4 <> sc2              ≡  true
```

* Operadores sobre *records*

The *record modification* has the form:

```
mu (e, id1 |-> e1, id2 |-> e2, ..., idn |-> en)
```

where the evaluation of the expression `e` returns the record value to be modified. All the identifiers `idi` must be distinct named entrances in the record type of `e`.

Examples: If `sc` is the value `mk_Score(<France>,3,0,0,9)` then

```
mu(sc, drawn |-> sc.drawn + 1, points |-> sc.points + 1)
≡ mk_Score(<France>,3,1,0,10)
```

Na realidade não modifica, gera um novo *record* modificado!

Operadores sobre tuplos

(Lembrar que tuplos são instâncias de produtos de tipos)

Constructors: The tuple constructor: `mk_(a1, a2, ..., an)`

Operators:

Operator	Name	Type
`t.#n`	Select	$T * \mathsf{nat} \rightarrow Ti$
`t1 = t2`	Equality	$T * T \rightarrow \mathsf{bool}$
`t1 <> t2`	Inequality	$T * T \rightarrow \mathsf{bool}$

Examples: Let `a = mk_(1, 4, 8)`, `b = mk_(2, 4, 8)` then:

```
a = b          ≡ false
a <> b         ≡ true
a = mk_(2,4)   ≡ false
```

Expressão "let"

- let *definição1, definição2, …* in *expressão*
  - Devolve valor da expressão da parte "in" usando definições de variáveis ou funções introduzidas na parte "let"
  - Definição de variável (no sentido matemático): identificador = expressão
  - Definição de variável (no sentido matemático) c/ padrão: padrão = expressão
  - Definição de função: usando sintaxe habitual para definir função de forma explícita (assinatura introduzida antes dos nomes dos argumentos)
  - Permite aumentar a legibilidade da especificação
- let *identificador* in set *conjunto* [be st *condição*] in *expressão*
  - Escolhe um elemento arbitrário do conjunto para usar na expressão
  - Devolve o valor da expressão da parte "in"
  - No caso da parte "be st" existir (ler "be such that"), escolhe um elemento obedecendo à condição
  - Outras formas: ver manual

Exemplos de "let"

- Exemplo: Função para obter o máximo dum conjunto de reais

```
max(s: set of real) res: real ==
   let  x in set s
   in   let resto = s \ {x}
        in  if resto = {} then x
            else let max2: real * real -> real
                     max2(a,b) == if a > b then a else b
                 in  max2(x,  max(resto))
   pre s <> {};
```

Escolhe valor arbitrário x de conjunto s para usar a seguir

Define variável resto para usar a seguir

Define função max2 para usar a seguir

```
max(s: set of real) res: real ==
   let x in set s be st (not exists y in set s & y > x)
   in x
   pre s <> {};
```

Escolhe valor de conjunto obedecendo a condição

Expressões "if" e "cases"

- if *condição* then *expressão1* else *expressão2*
  - Se a *condição* for verdadeira, dá o valor da *expressão1*, senão dá o valor da *expressão2*
  - **"elseif"** = "else if"
- cases *expressão*:
  padrão11, ..., padrão1N -> expressão1,
  ... ,
  padrãoM1, ..., padrãoMN -> expressãoM,
  others -> *expressãoM1*
  end
  - Os padrões (ver a seguir) são comparados por ordem com a *expressão*
  - Se o 1º padrão a fazer "match" é *padrãoij* , devolve o valor de *expressãoi*
  - Parte "others" é opcional
  - Semelhante a "switch", com a diferença de se poderem usar padrões

Exemplo de "cases" com constantes

Caso simples em que os padrões são constantes literais

```
functions

  public static DaysOfMonth(year, month: nat1) res : nat1 ==
  (
    cases month :
      1, 3, 5, 7, 8, 10, 12 -> 31,
      4, 6, 9, 11 -> 30,
      2 -> if IsLeapYear(year) then 29 else 28
    end
  )
  pre month >= 1 and month <= 12;
```

Exemplo de "cases" com padrões

Caso de sequência vazia (usa padrão de constante literal)

Caso de sequência com um único elemento x (usa padrão de enumeração de sequência e padrão de identificador)

Caso de sequência que se pode exprimir como a concatenação de duas subsequências não vazias (usa padrão de concatenação de sequências e padrão de identificador)

Usa padrão "valor de expressão". Necessita parêntesis!

```
functions

mergesort (s: seq of nat) res: seq of nat ==
  cases s:
    [] -> [],
    [x] -> [x],
    s1 ^ s2 -> merge (mergesort(s1), mergesort(s2))
  end;

merge(s1, s2: seq of nat) res : seq of nat ==
  cases true:
    (s1 = []) -> s2,
    (s2 = []) -> s1,
    (hd s1 < hd s2) -> [hd s1] ^ merge(tl s1, s2),
    others -> [hd s2] ^ merge(s1, tl s2)
end;
```


Padrões

Tipo de padrão	Descrição
identifier	Faz *match* com qualquer valor, fixando o valor do identificador.
literal	Constante literal. Só faz *match* com o mesmo valor.
-	*Don't care*.
(*expression*)	Valor de uma expressão. Só faz *match* com o mesmo valor. Parêntesis podem ser necessários para não confundir com outros padrões.
{p1, p2, …}	Faz *match* com um conjunto com o mesmo nº de elementos, e que fazem *match* com os padrões p1, p2, etc. (por qualquer ordem).
p1 union p2	Faz *match* com um conjunto que se pode exprimir como a união de dois subconjuntos disjuntos não vazios que fazem *match* com os padrões p1 e p2.
[p1, p2, …]	Faz *match* com uma sequência com o mesmo nº de elementos, e que fazem *match* com os padrões p1, p2, etc. (pela mesma ordem).
p1 ^ p2	Faz *match* com uma sequência que se pode exprimir como a concatenação de duas subsequências não vazias que fazem *match* com os padrões p1 e p2.
mk_(p1, p2, …)	Faz *match* com um tuplo com o mesmo nº de componentes, e que fazem *match* com os padrões indicados (pela mesma ordem).
mk_*RecordName*(p1, p2, …)	Faz *match* com um *record* do mesmo tipo e com o mesmo nº de componentes, e que fazem *match* com os padrões indicados (pela mesma ordem).

Quantificadores

- forall *binding* & *condição* ∀ "qualquer que seja … verifica que …"
- exists *binding* & *condição* ∃ "existe pelo menos um … tal que …"
- exists1 *binding* & *condição* $\exists^1$ "existe um e um só … tal que …"

Ligações (*bindings*)

- *Set binding*: *padrão1, ..., padrãoN* in set *conjunto*
  - Tenta fazer *match* dos padrões com os elementos do conjunto, fixando (*binding*) os valores dos identificadores introduzidos nos padrões
  - Na maioria dos casos, os padrões são simplesmente identificadores
- *Type binding*: *padrão1, ..., padrãoN* : *tipo*
  - Semelhante ao anterior, só que considera as instâncias de um tipo (potencialmente em nº infinito!) em vez dos elementos de um conjunto
  - *Type bindings* não são executados pelo interpretador, pois um tipo de dados pode ter um número infinito de instância!
- Usados principalmente em quantificadores e definição de colecções em compreensão

Exemplos com quantificadores

- Exemplo: restrição de unicidade de chave

```
-- número é chave de sócio, isto é, não podem existir
-- dois sócios com o mesmo número
inv not exists s1, s2 in set sócios &
          s1 <> s2 and s1.número = s2.número;
```

ou, equivalentemente (pelas leis de De Morgan):

```
inv forall s1, s2 in set sócios  &
        s1 <> s2  =>  s1.número <> s2.número;
```


Expressão de selecção "iota"

- iota *identificador* in set *conjunto* & *condição*
- Selecciona o único elemento do conjunto que obedece à condição
- Outras formas da expressão "iota": ver manual
- Exemplo: Operação de selecção de sócio pelo número (chave)

```
public GetSócio(número: nat1) res: Sócio ==
  return iota s in set sócios  &  s.GetNúmero() = número
pre exists1 s in set sócios & s.GetNúmero() = número;
```

- Exemplo do máximo

```
max(s: set of real) res: real ==
  iota x in set s & (not exists y in set s & y > x)
pre s <> {};
```

Teste de pertença a tipo

Sintaxe	Semântica	Exemplos
is_*typeName*(*expression*)	Verifica se a expressão é do tipo indicado. Só é aplicável a tipos básicos e *record*s.	is_bool(1) ≡ false is_Date(mk_Date(2001,1,1)) ≡ true
is_(*expression*, *typeExpr*)	Verifica se a expressão é do tipo indicado. É aplicável também a outros tipos.	size: set of nat | seq of nat -> nat size(s) == if is_(s, seq of char) then len s else card s;

Útil sobretudo quando se usam uniões!

Teste de pertença a classe

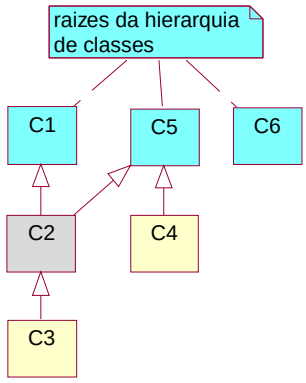
Sintaxe	Semântica
isofclass(*className*, *object_ref*)	Verifica se o objecto indicado é uma instância da classe indicada ou de uma subclasse
sameclass(*object_ref1*, *object_ref2*)	Verifica se os dois objectos são instâncias da mesma classe
isofbaseclass(*className*, *object_ref*)	Verifica se o objecto indicado é instância duma classe que tem a classe indicada como raiz na hierarquia de classes
samebaseclass(*object_ref1*, *object_ref2*)	Verifica se os dois objectos são instâncias de classes que têm uma classe comum como raiz na hierarquia de classes

Útil sobretudo quando se usam subclasses (equivalente OO a uniões)!

Teste de pertença a classe

Considerando Oi instância de Ci, em relação a O2 são verdadeiras (apenas) as seguintes expressões:

isofclass(C2,O2)
isofclass(C1,O2)
isofclass(C5,O2)
isofbaseclass(C1,O2)
isofbaseclass(C5,O2)
sameclass(O2,O2)
samebaseclass(O1,O2)
samebaseclass(O2,O2)
samebaseclass(O3,O2)
samebaseclass(O4,O2)
samebaseclass(O5,O2)



Anexo C: Funções avançadas em VDM++

Funções avançadas

- Funções polimórficas
- Funções de ordem superior
- O tipo função
- A expressão lambda

Funções polimórficas

- nomeFunção[@*TypeParam1*, @*TypeParam2*, …] …

- São funções genéricas, que podem ser usadas com valores de diferentes tipos
- Têm parâmetros especiais (*type parameters*) que devem ser substituídos por nomes de tipos concretos ao usar a função
- Nomes desses parâmetros começam por "@" e são indicados entre parêntesis rectos a seguir ao nome da função
- Semelhantes a *function templates* em C++

Funções polimórficas

- Exemplo: função utilitária, que verifica se uma sequência de elementos de um tipo qualquer tem duplicados:

```
public static HasDuplicates[@T](s: seq of @T) res: bool ==
      exists i, j in set inds s & i <> j and s(i) = s(j);
```

- Exemplo de utilização dessa função:

```
class Publicação
 instance variables
   private autores: seq of Autor := [];
   inv not HasDuplicates[Autor](autores);
```

Passa um tipo concreto

Funções de ordem superior

- São funções que recebem outras funções como argumentos, ou (*Curried functions*) que retornam funções como resultado
- I.e. têm argumentos ou resultado do tipo função
- Exemplo: função que acha um zero aproximado duma função entre limites especificados, com erro máximo especificado, pelo método das bissecções sucessivas:

```
findZero( f: real -> real , x1, x2, err: real) res: real ==
  if abs(x1 - x2) <= err and abs(f(x1) – f(x2)) <=  err then x1
  else let m = (x1 + x2) / 2
       in if sinal(f(m)) = sinal(f(x1)) then findZero(m,x2)
          else findZero(x1,m)
 pre sinal(f(x1)) <> sinal(f(x2));
```

O tipo função

- Função total: *arg1Type * arg2Type * …* **+>** *resultType*
- Função parcial: *arg1Type * arg2Type * …* **->** *resultType*

Pode-se ver como um único argumento do tipo tuplo
(instância do produto de tipos)

- O tipo de uma função é definido pelos tipos de argumentos e resultado
- Instâncias de um tipo de função (i.e. funções concretas) podem ser passadas como argumento ou retorno de funções,
e guardadas (por referência) em estruturas de dados

O tipo função

Operator	Name	Type
`f1 comp f2`	Function composition	$(B \to C) * (A \to B) \to (A \to C)$
`f ** n`	Function iteration	$(A \to A) * \mathsf{nat} \to (A \to A)$

Operator Name	Semantics Description
Function composition	it yields the function equivalent to applying first `f2` and then applying `f1` to the result. `f1`, but not `f2` may be Curried.
Function iteration	yields the funciton equivalent to applying `f n` times. `n=0` yields the identity function which just returns the value of its parameter; `n=1` yields the function itself. For `n>1`, the result of `f` must be contained in its parameter type.

A expressão lambda

- lambda *padrãoArg1*: *Tipo1*, …, *padrãoArgN*: *TipoN* & *expr*

λ

- Constrói uma função *on the fly*
- Padrões normalmente são simplesmente identificadores de argumentos
- Usado normalmente para passar como argumento a outra função (de ordem superior)
- Exemplo: achar um zero real de um polinómio

```
findZero(lambda x: real & 5 * x**3 -  x**2 - 2 , 0, 1, 0.0000001)
```


Anexo D: Exemplo da Agenda Corporativa

Caracterização sumária do sistema

- Cada recurso (pessoa, espaço ou equipamento) tem uma agenda associada (*date book*), com compromissos (*appointments*) e disponibilidades (*slots*)
- Um compromisso pode envolver vários recursos e pode ocupar um conjunto de intervalos de tempo
- Os compromissos marcados por terceiros estão sujeitos às disponibilidade manifestadas
- O sistema deve ajudar o utilizador a marcar compromissos, mostrando hipóteses de marcação
- Exemplos de aplicações: marcação de consultas, marcação de reuniões, reserva de equipamentos, etc.

Requisitos (1)

1. O sistema deve permitir gerir a marcação de compromissos envolvendo vários recursos numa organização.
2. Os recursos podem ser pessoas, espaços ou equipamentos.
3. Cada recurso tem uma agenda associada (*datebook*).
4. Cada agenda tem um dono, que é uma pessoa da organização.
5. O dono da agenda de uma pessoa é a própria pessoa.
6. Uma agenda (*datebook*) tem elementos de dois tipos: compromissos (*appointments*) e disponibilidades (*slots*).
7. Um compromisso é de um certo tipo (reunião, aula, etc.), envolve um ou mais recursos e ocupa um intervalo de tempo ou um conjunto de intervalos de tempo, com resolução ao minuto.
8. Deve ficar registado quem marcou um compromisso (autor do compromisso).

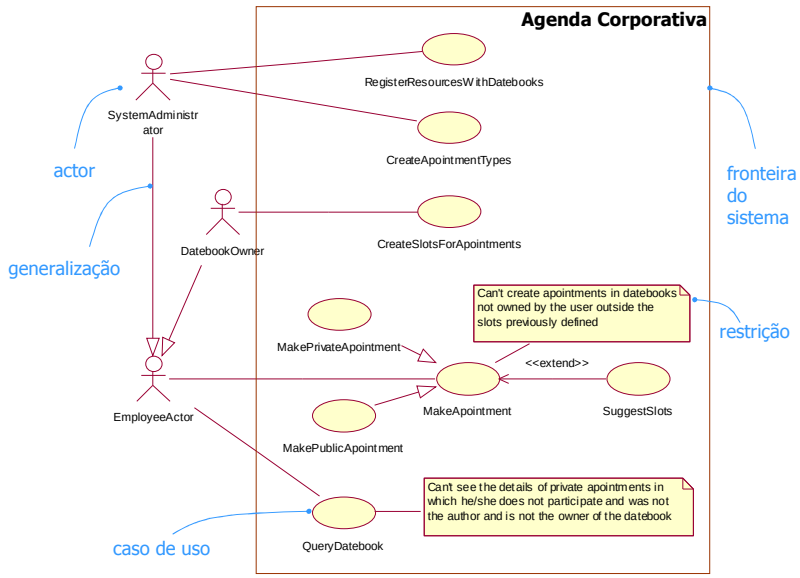
Requisitos (2)

9. Um recurso não pode ter dois compromissos ao mesmo tempo.
10. Uma disponibilidade refere-se a um recurso, um tipo de compromisso e um intervalo de tempo ou um conjunto de intervalos de tempo.
11. Quando a pessoa só pode marcar um compromisso numa agenda de que não é dona, dentro das disponibilidades definidas pelo dono da agenda. Só o dono da agenda pode marcar compromissos sem verificação de disponibilidades.
12. As agendas podem ser consultadas por todas as pessoas da organização.
13. Os compromissos podem ser públicos ou privados. Um compromisso privado só pode ser consultado pelo autor do compromisso, pelas pessoas envolvidas no compromisso e pelos donos das agendas dos recursos envolvidos.

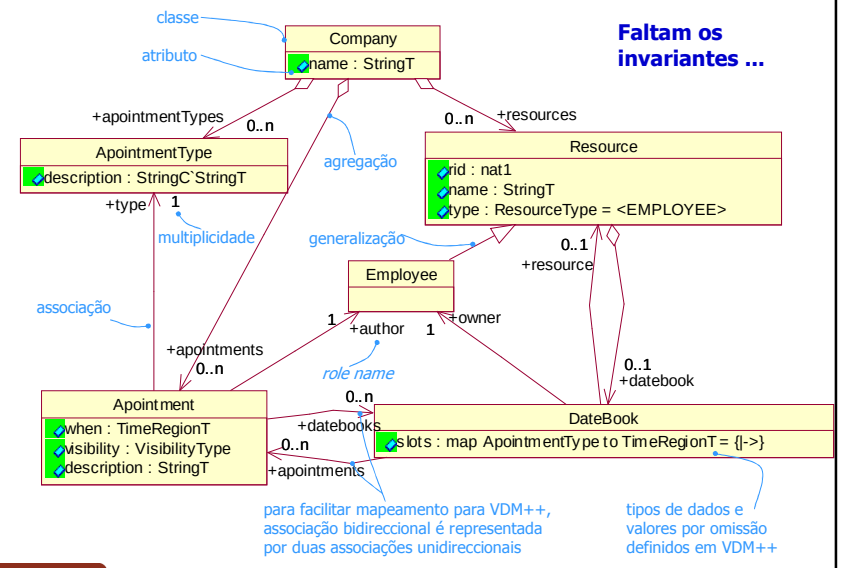
Requisitos (3)

14. O sistema deve ajudar o utilizador a marcar compromissos da seguinte forma: o utilizador indica o tipo de compromisso a marcar (exemplo: reunião), os recursos envolvidos, a duração do compromisso e restrições temporais para a marcação do compromisso; o sistema deve fornecer uma lista de hipóteses de marcação; cada hipótese mostra um intervalo de tempo dentro do qual é possível marcar o compromisso em todos os recursos (porque têm declarada disponibilidade para o tipo de compromisso especificado e não têm marcados compromissos de qualquer tipo).
15. Compete ao administrador do sistema registar os recursos da organização (criando automaticamente as respectivas agendas) e definir os tipos de compromissos possíveis.

Modelo de casos de uso em UML



Modelo de domínio em UML - estado

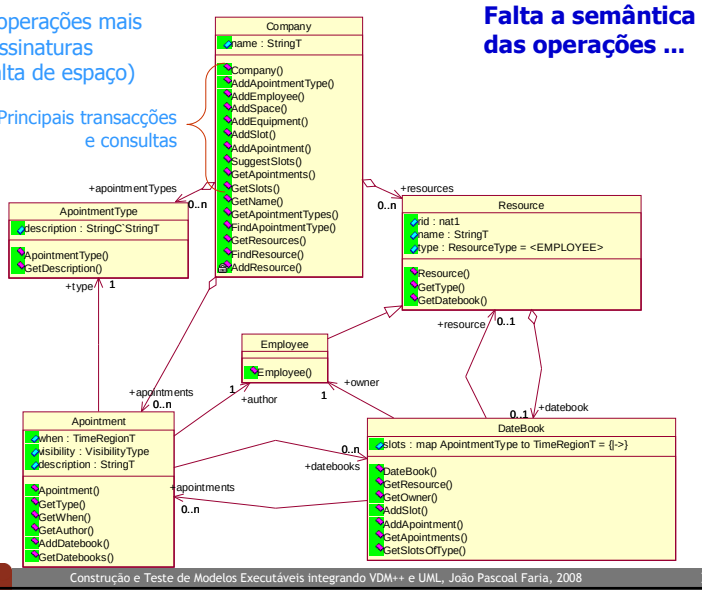


Modelo de domínio em UML - operações

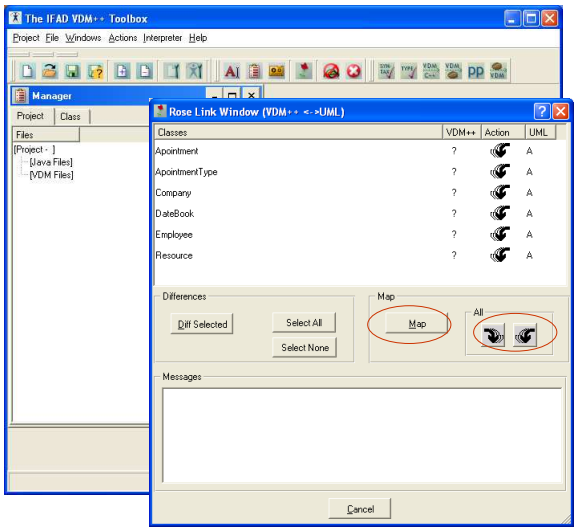
Agora com as operações mais importantes (assinaturas omitidas por falta de espaço)

Principais transacções e consultas

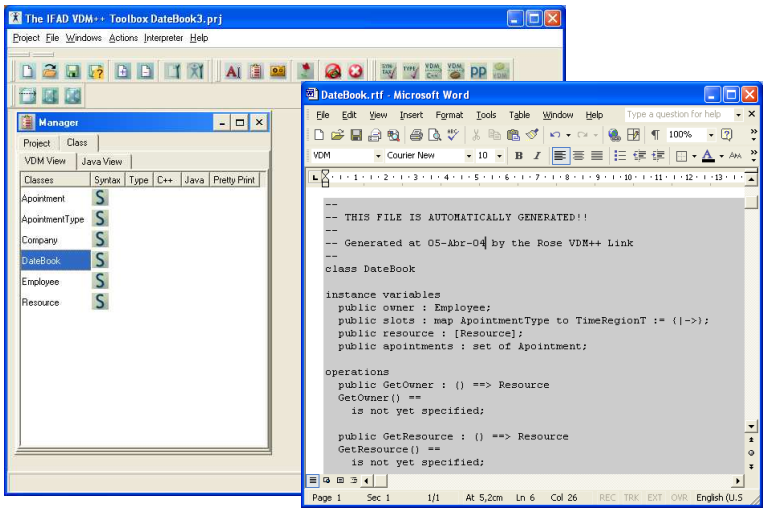
Falta a semântica das operações ...



Tradução de UML para VDM++ (1)



Tradução de UML para VDM++ (2)



Definição de tipos de dados em VDM++ (1)

- Tipos de dados a definir neste caso (*):

Tipo	Descrição
StringT	cadeia de um ou mais caracteres
Date	data representada pelo número de dias decorridos desde uma data base, útil para processamento interno
UserDate	data representada por uma combinação de dia, mês e ano
Time	instante de tempo, representado pelo número de minutos decorridos desde as 0 horas da data base
UserTime	instante de tempo, representado por uma combinação de minutos, horas, dia, mês e ano
TimeInterval	intervalo entre dois instantes de tempo, fechado à esquerda e aberto à direita, representado por 2 valores do tipo Time
Duration	uma duração em minutos

(*) Normalmente estes tipos de dados estariam definidos numa biblioteca

Definição de tipos de dados em VDM++ (2)

- Tipos de dados a definir neste caso (cont.)

Tipo	Descrição
UserTimeInterval	intervalo entre dois instantes de tempo, fechado à esquerda e aberto à direita, representado por um par de valores do tipo UserTime
TimeRegion	**região de tempo (conjunto de intervalos de tempo) representada por um conjunto de intervalos de tempo unitários (com 1 min de duração), representados pelos seus inícios (valores do tipo Time)** representação abstracta pouco eficiente, mas que permite efectuar operações com regiões de tempo usando directamente os operadores de conjuntos (reunião, intersecção e diferença), simplificando a especificação
ResourceType	tipo enumerado que admite os valores <EMPLOYEE>, <EQUIPMENT> ou <SPACE>
VisibilityType	tipo enumerado que admite os valores <PUBLIC> ou <PRIVATE>

Exemplo de definição de tipos de dados em VDM++

```
types
  public UserDate :: year : nat1
                     month: nat1
                     day  : nat1
  inv d == d.month <= 12 and d.year >= Year0
           and d.day <= DaysOfMonth(d.year, d.month);
values
  private Year0 : nat1 = 1900;
  private UserDate0 : UserDate = mk_UserDate(Year0,1,1);

functions
  private static IsLeapYear(year: nat1) res : bool ==
    year mod 4 = 0 and year mod 100 <> 0 or year mod 400 = 0;

  private static DaysOfMonth(year, month: nat1) res : nat ==
    (cases month :
      1, 3, 5, 7, 8, 10, 12 -> 31,
      4, 6, 9, 11 -> 30,
      2 -> if IsLeapYear(year) then 29 else 28
     end)
  pre month <= 12;
(...)
```

definição de tipo de *record*

invariante

constantes

construção de *record*

pré-condição

Exemplo de definição de classe em VDM++ (variáveis de instância)

```
class Company

instance variables
  public name : StringT;

  public apointmentTypes : set of ApointmentType := {};
   -- Must be initialized because of invariant.

  public resources : set of Resource := {};
   -- Must be initialized because of invariant.

  public apointments : set of Apointment := {};
   -- Because there is a many to many relationship between
   -- datebooks and apointments, they "belong" to the company
   -- and are referenced by datebooks (and vice versa)
...
```

só para facilitar testes

conjunto de referências para objectos

Exemplo de definição de classe em VDM++ (invariantes)

```
...
  -- Key constraints
  -- uniqueness of Resource.rid
  inv forall r1, r2 in set resources &
          r1<> r2 => r1.rid <> r2.rid;
   -- Limitation: not checked if r.rid is changed
...

  -- Referential integrity constraints
  -- referential integrity of Apointment.type
  inv forall a in set apointments &
          a.type in set apointmentTypes;
   -- Limitation: not checked if a.type is changed
...

   -- Generic constraints
...
```

Exemplo de definição de classe em VDM++ (operações de alteração)

```
operations

public AddApointmentType(desc: StringT)
          res : ApointmentType ==
(
  dcl t: ApointmentType := new ApointmentType(desc);
  apointmentTypes := apointmentTypes union {t};
  return t
)
ext wr apointmentTypes
pre desc not in set {at.description | at in set apointmentTypes}
post apointmentTypes = apointmentTypes~ union {res}
        and res.description = desc;
...
```

variável de instância que a operação pode alterar

pré-condição implicada por invariante (pode-se efectuar verificação formal)

"~" designa valor antigo da variável

Exemplo de definição de classe em VDM++ (operações de consulta)

```
public SuggestSlots(type: ApointmentType, rs: set of Resource,
          duration: TimeDurationT, within: TimeRegionT)
          res: set of UserTimeIntervalT ==
 return
 {TimeIntervalToUserTimeInterval(ti) |
     ti in set
        TimeRegionToTimeIntervals(
           dinter {
              let r_slots = r.GetDatebook().GetSlotsOfType(type),
                  r_apoints = dunion {a.when | a in set
                                      r.GetDatebook().GetApointments()}
              in (within inter r_slots \ r_apoints)
            | r in set rs
           }
        )
     & Duration(ti)>=duration
 };
```

{f(x) | x in set ...} – define conjunto em compreensão
dinter {...} – intersecção de elementos de conjunto
dunion {...} – união de elementos de conjunto
& - tal que

Anexo E: Exemplo da Colocação de Professores

Problema

- Portugal, 2004
- Professores concorrem a vagas; dois tipos de professores:
  - tinham colocação, mas pretendem mudar (se não for possível, ficam onde estavam)
  - não tinham colocação, e pretendem obtê-la (se não for possível, ficam sem colocação)
- Cada professor indica lista totalmente ordenada de vagas a que concorre
- Vagas a concurso incluem posições anteriormente ocupadas pelos professores que pretendem mudar de colocação
- Os professores já estão totalmente ordenados segundo um *ranking*
- Neste *ranking*, podem aparecer intercalados professores dos dois tipos
- O resultado da colocação deve obedecer às seguintes restrições:
  - R1: Professores que tinham colocação anterior têm de ficar colocados, nem que seja nessa posição
  - R2: Para cada professor e para cada posição por ele preferida em relação àquela em que foi colocado (inclui todas as posições no caso de ficar por colocar), essa posição tem de estar ocupada por um professor com melhor ranking ou pelo professor que aí estava anteriormente

Formalização do problema (1)

```
public static colocaProfessores(vagas: set of Vaga, professores: seq of Professor,
preferencias: map Professor to seq of Vaga, colocacaoInicial: inmap Professor to Vaga)
colocacaoFinal: inmap Professor to Vaga
 -- vagas: conjunto de vagas a concurso, incluindo posições ocupadas inicialmente por
 --          professores que querem mudar
 -- professores: conjunto de professores, ordenado por ranking
 -- preferencias: a cada professor faz corresponder uma lista ordenada de vagas
 --                  pretendidas por ordem decrescente
 -- colocacaoInicial: indica os professores que tinham colocação inicial e qual a sua
 --                       colocação inicial
 -- colocacaoFinal: colocação final dos professores
pre
    -- P1: não existem professores repetidos
    not hasRepetitions[Professor](professores) and

    -- P2: todos os professores indicam preferências
    (dom preferencias = elems professores) and

    -- P3: as vagas referidas nas preferências estão no conjunto das vagas
    (forall p in set dom preferencias & elems preferencias(p) subset vagas) and
```


Formalização do problema (2)

```
  -- P4: a lista de preferências de um professor não tem repetições
  (forall p in set dom preferencias & not hasRepetitions[Vaga](preferencias(p))) and

  -- P5: os professores com colocação inicial estão no conjunto dos professores
  (dom colocacaoInicial subset elems professores) and

  -- P6: as vagas referidas na colocação inicial estão no conjunto das vagas
  -- (implicado por P3 e P7)
  (rng colocacaoInicial subset vagas) and

  -- P7: no caso de professores que tinham colocação inicial, a sua colocação inicial
  -- deve estar no fim da lista de preferências
  (forall p in set dom colocacaoInicial &
       colocacaoInicial(p) = preferencias(p)(len preferencias(p)))

post
    -- Q1: os professores referidos na colocação final têm de estar no conjunto de
    -- professores
    (dom colocacaoFinal subset elems professores) and

    -- Q2: as vagas referidas na colocação final têm de estar no conjunto de vagas
    (rng colocacaoFinal subset vagas) and
```

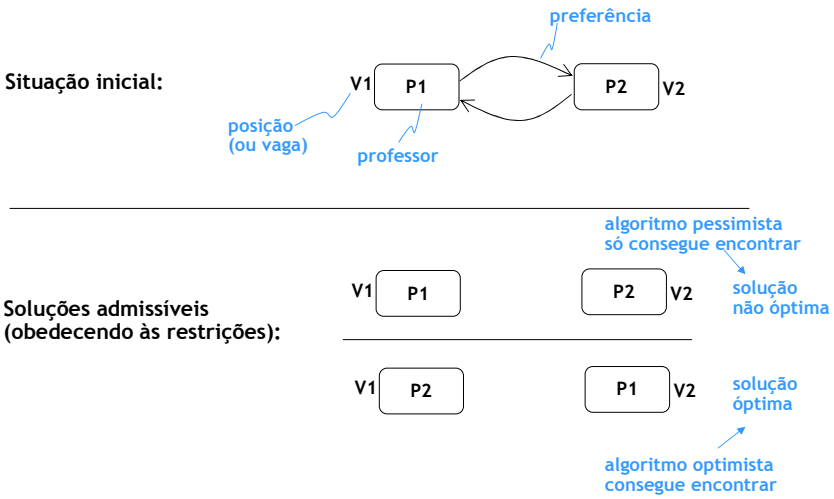
Formalização do problema (3)

```
-- Q3: um professor só pode ser colocado numa vaga que está nas suas preferências
(forall p in set dom colocacaoFinal &
      colocacaoFinal(p) in set elems preferencias(p)) and

-- Q4: os professores que estavam inicialmente colocados têm de continuar colocados
-- no fim  (implicado por Q5, dada a pré-condição P7 e a forma como está definida a
-- função "temPrecedencia")
(dom colocacaoInicial subset dom colocacaoFinal) and

-- Q5: todas as vagas pretendidas por um professor, com maior preferência em
-- relação à posição em  que ficou, têm de estar ocupadas por alguém com
-- precedência sobre essa vaga (tem precedência 1º quem estava aí colocado e depois
--  segue-se o ranking) estava colocado inicialmente
(forall p in set elems professores &
  let indiceColocacao = if p in set dom colocacaoFinal then
                        index[Vaga](colocacaoFinal(p), preferencias(p))
                       else
                        len preferencias(p) + 1
   in forall i in set {1, ... , indiceColocacao - 1} &
       let v = preferencias(p)(i)
       in  v in set rng colocacaoFinal and
          temPrioridade((inverse colocacaoFinal)(v),p, v, professores, colocacaoInicial));
```


Critério de optimalidade

- Pode existir mais do que uma solução que satisfaz as pós-condições
- Designando soluções admissíveis a todas as soluções que satisfazem as restrições indicadas, introduz-se o seguinte critério de optimalidade:
  - Uma solução admissível S1 é **melhor** do que uma solução admissível S2, se o 1º professor por ordem de *ranking* que tem colocação diferente nas duas soluções, está melhor colocado em S1
- Por este critério, só há uma só solução óptima, isto é, que é melhor que todas as outras

Exemplo com várias soluções admissíveis



Algoritmos

- Algoritmo de "força bruta"
  - Gerar o conjunto de todos os mapeamentos injectivos de professores para vagas, rejeitar depois os que não obedecem às pós-condições e finalmente aplicar o critério de optimalidade
  - Tempo exponencial
- Algoritmo "pessimista"
  - Seguido pela Compta, empresa que não conseguiu resolver o problema em 2004
  - Tempo polinomial, não garante solução óptima (ver 1º exemplo)
- Algoritmo "optimista"
  - Desenvolvido pela ATX Software, empresa que resolveu o problema em 2004
  - Tempo polinomial, parece garantir solução óptima
- Algoritmo de Gale-Shapley (dos casamentos estáveis)
  - Mais eficiente (pelo menos em teoria), garante solução óptima (?)

Algoritmo de colocação pessimista

1. Consideram-se inicialmente ocupadas as posições iniciais dos professores que pretendem mudar de posição
2. Colocam-se os professores pela ordem do ranking:
   - 2.1. Coloca-se o professor na sua melhor preferência que está ainda livre
   - 2.2. Se o professor estava inicialmente colocado e é colocado agora numa nova posição, gera uma vaga que tem de ser recuperada:
     - 2.3.1. Procura-se o professor já colocado de melhor ranking que pode beneficiar dessa vaga
     - 2.3.2. Se não se encontrar nenhum professor nessas condições, a recuperação da vaga fica concluída
     - 2.3.3. Se se encontrar um professor nessas condições, muda-se a colocação desse professor (melhorando-a), o que origina uma nova vaga que tem de ser recuperada da mesma forma

Algoritmo de colocação optimista (*)

1. Consideram-se inicialmente livres as posições iniciais dos professores que pretendem mudar de posição
2. Colocam-se os professores pela ordem do ranking, na melhor preferência ainda livre de cada professor (alguns professores podem ficar por colocar)
3. Se os professores que estavam inicialmente colocados ficaram todos colocados, o processo termina.
4. Senão,
   - 4.1. Os professores que estavam inicialmente colocados e ficaram por colocar são colocados definitivamente nas suas posições iniciais, que deixam de estar livres
   - 4.2. Repete-se a colocação com menos estes lugares livres

(*) Desenvolvido pela ATX SOFTWARE

Relação com problema de casamentos estáveis

- Problema de casamentos estáveis, versão clássica:
  - Supondo que cada elemento dum grupo de *n* homens e *n* mulheres ordenou todos os de sexo oposto por ordem de preferência estrita, pretende-se determinar um emparelhamento estável.
  - Sendo *H = {h1, . . . , hn}* e *M = {m1, . . . ,mn}* os conjuntos de homens e mulheres, um emparelhamento *E* é uma qualquer função injectiva de *H* em *M*. Informalmente, um emparelhamento é, neste caso, um conjunto de *n* casais (monogâmicos e heterossexuais).
  - Um emparelhamento *E* diz-se instável se e só se existir um par *(h,m)* $\notin$ *E* tal que *h* prefere *m* à sua parceira em *E* e *m* também prefere *h* ao seu parceiro em *E*. Caso contrário, diz-se estável.

Relação com problema de casamentos estáveis

- Problema de casamentos estáveis com listas de preferências incompletas
  - Surgiu na colocação de internos em hospitais
  - O critério de estabilidade das soluções é reformulado do modo seguinte: Um emparelhamento é instável se e só se existir um candidato r e um hospital h tais que h é aceitável para r e r é aceitável para h, o candidato r não ficou colocado ou prefere h ao seu actual hospital e h ficou com vagas por preencher ou h prefere r a pelo menos um dos candidatos com que ficou. Caso contrário, diz-se estável.

Relação com problema de casamentos estáveis

ALGORITMO DE GALE-SHAPLEY (ORIENTADO POR INTERNOS) **(1962)**

Considerar inicialmente que todos os internos estão livres.
Considerar também que todas as vagas nos hospitais estão livres.
Enquanto existir algum interno r livre cuja lista de preferências é não vazia
    seja h o primeiro hospital na lista de r;
    se h não tiver vagas
        seja r' o pior interno colocado provisoriamente em h;
        r' fica livre (passa a não estar colocado);
    colocar provisoriamente r em h;
    se h ficar sem vagas então
        seja s o pior dos colocados provisoriamente em h;
        para cada sucessor s' de s na lista de h
            remover s' e h das respectivas listas
fim

Relação com problema de casamentos estáveis

- Propriedades do algoritmo de Gale-Shapley (1962):
  - O emparelhamento obtido por este algoritmo é óptimo para os internos e péssimo para os hospitais: qualquer interno fica com o melhor hospital que pode ter em qualquer emparelhamento estável e cada hospital fica com os piores internos.
  - Tempo de execução é de ordem quadrática (nº de internos * nº de hospitais)

Relação com problema de casamentos estáveis

- No problema da colocação de professores, o conceito de emparelhamento é o mesmo, se se considerar que a atribuição é de candidatos a vagas
- Internos correspondem aos professores
- Hospitais correspondem às vagas (ou escolas?)
- Cada vaga prefere 1º o professor que aí estava colocado anteriormente, e depois todos os outros pela ordem do ranking

Formalização do algoritmo (corpo)

```
dcl colocacaoProvisoria: inmap Professor to Vaga := {|->};
dcl prefsRestantes: map Professor to seq of Vaga := preferencias;
dcl profsColocaveis: seq of Professor := professores;
while profsColocaveis <> [] do
  -- escolhe um professor colocável (p) e a vaga (v) à cabeça da sua lista de preferências
  -- (esolhe o 1º professor por ordem de ranking, mas não era obrigatório)
  let p = hd profsColocaveis, v = hd prefsRestantes(p) in (
    -- coloca o professor escolhido na vaga preferida, tirando quem aí estava eventualmente
    colocacaoProvisoria := (colocacaoProvisoria :-> {v}) munion {p |-> v};
    -- tira esta vaga das listas de preferências de todos os professores candidatos a esta
    -- vaga, mas com menor prioridade que o professor seleccionado
    for all p2 in set dom prefsRestantes do
      if v in set elems prefsRestantes(p2) then
        if temPrioridade(p, p2, v, professores, colocacaoInicial) then
          prefsRestantes(p2) := remove[Vaga](v, prefsRestantes(p2));
    -- recalcula os professores colocáveis (por colocar, c/lista de preferências não vazia)
    profsColocaveis := [professores(i) | i in set inds professores &
                        professores(i) not in set dom colocacaoProvisoria
                        and prefsRestantes(professores(i)) <> [] ]
);
return colocacaoProvisoria
```

Casos de teste em ColocacaoProfessores.zip