Desenvolvimento Guiado por Modelos (MDD) usando Métodos Formais e UML

João Pascoal Faria

Seminário de Sistemas de Informação
Instituto Politécnico de Viana do Castelo
Escola Superior de Tecnologia e Gestão
14/3/2007

Agenda

- Desenvolvimento guiado por modelos (MDD)
- Dos modelos visuais aos modelos formais, executáveis e traduzíveis
- Exemplo
- Conclusões

Desenvolvimento guiado por modelos (MDD)

What is model-driven software engineering?

- ***Model-Driven Engineering*** *(or MDE) refers to the systematic use of models as primary engineering artifacts throughout the engineering lifecycle. MDE can be applied to software, system, and data engineering. Models are considered as first class entities.*
- ***Model-Driven Software Engineering*** (or MDSE) is the application of MDE to software engineering, i.e., it consists of systematically using models as primary engineering artefacts throughout the software engineering lifecycle.

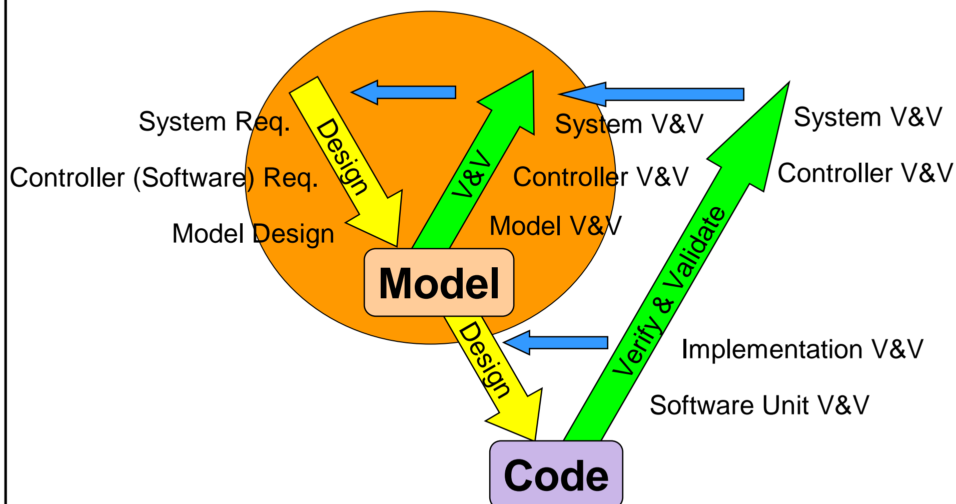
What is a model?

- *Modeling, in the broadest sense, is the cost-effective use of something in place of something else for some cognitive purpose. It allows us to use something that is simpler, safer or cheaper than reality for some purpose. A model represents reality for the given purpose; the model is an* ***abstraction*** *of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the* ***complexity****, danger and irreversibility of reality.*
  - "The Nature of Modeling in Artificial Intelligence, Simulation, and Modeling", John Wiley and Sons, 1989, pp. 75-92
- **A model of a software system** is a simplified abstract representation of a software system. The most common types of models use today are visual models (based on diagrams) and formal models (based on mathematics)

MDE @

"We use **modeling** and **simulation** for early validation of the system solution"

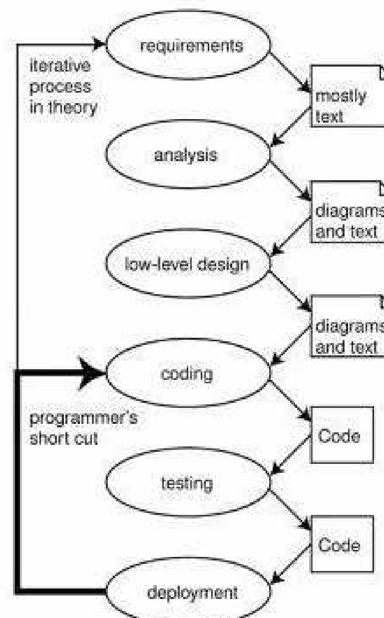
Tim Davis, A-MOST'06



Model-Driven Architecture

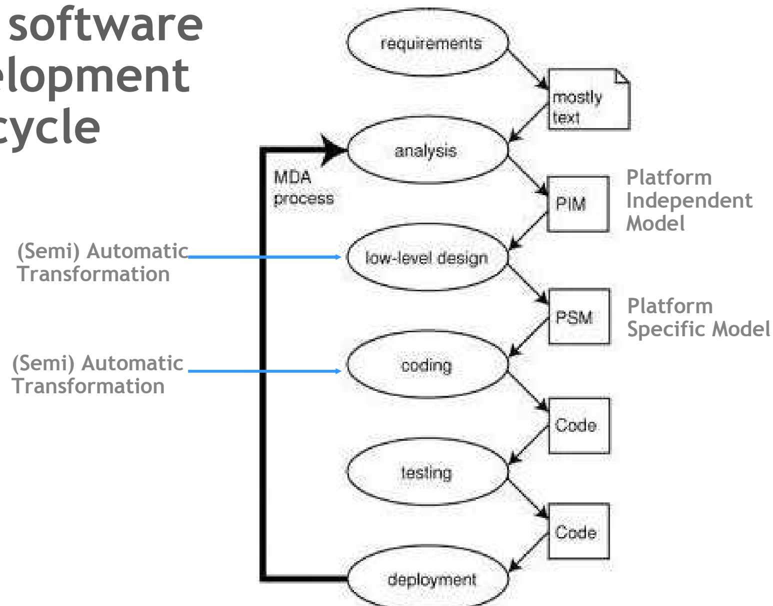
- The best known MDE initiative is the Object Management Group (OMG) initiative called Model-Driven Architecture (MDA).
- Another related acronym from OMG is Model-Driven Development (MDD).
- MDA is intended to support model-driven engineering of software systems.
- **The basic idea behind the MDA initiative is to use (UML) models not only as analysis and design documents but also as the basis for code generation.**

Traditional software development life cycle



In "MDA Explained: The Model Driven Architecture -- Practice and Promise", by Anneke Kleppe, Jos Warmer and Wim Bast, from Addison-Wesley, 2003.

MDA software development life cycle



(Semi) Automatic Transformation

(Semi) Automatic Transformation

MDA benefits

- Increased productivity (with automated transformations from models to code as well as between models at different levels of abstraction)
  - **In order to be able to perform those transformations in an automated way, the models have to be of greater rigor and level of detail**
- Portability (with the distinction between Platform Independent Models and Platform Specific Models)
- Interoperability (via PSM bridges)
- Maintainability (models are easier to maintain than code)

Dos modelos visuais aos modelos formais, executáveis e traduzíveis

Modelos visuais

- Consensual e prática corrente:
  - Modelação visual (semi-formal) com diagramas UML durante as fases de análise e especificação de requisitos, desenho de alto nível (arquitectura) e desenho detalhado
  - Geração de algum código (Java, C#, SQL, XSD, etc.) a partir de UML e vice-versa (normalmente esqueletos de código, e não código completamente funcional)
- Modelos visuais são muito úteis para compreender e visualizar um sistema
- Modelos visuais 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)**

Dos modelos visuais aos 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) 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
    - Pode-se verificar a consistência interna da especificação
    - Podem-se gerar automaticamente casos de teste a partir da especificação, para verificar a conformidade duma implementação com a especificação

Dos modelos visuais aos 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 (em termos de capacidades) de uma linguagem de acções de alto nível
  - Infelizmente, a linguagem concreta não é fixada pela norma
- Obtém-se um modelo executável (com a ajuda de ferramentas), que serve como **protótipo executável** do sistema, permitindo testar e validar precocemente o sistema (validar requisitos funcionais e opções de *design*)

Dos modelos visuais aos modelos traduzíveis

- É de esperar que os modelos executáveis sejam também traduzíveis (com a ajuda de ferramentas) para uma linguagem de implementação-alvo (Java, C++, C#, ...)
- Estamos a falar de geração automática de código completamente funcional e não só esqueletos de classes a partir de modelos de alto nível
- Particularidades das linguagens, tecnologias e plataformas-alvo são embebidas nos geradores (conceito MDA – Model Driven Architecture)
- Vantagens: Aumento da produtividade no desenvolvimento de software, foco no domínio do problema e não nas tecnologias de implementação
- Desvantagens: Código gerado pouco eficiente, dificuldade de integração com bibliotecas existentes

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 (Object Constraint Language) - e modelos executáveis e traduzíveis - caso de xtUML - mas ainda não de forma perfeitamente integrada
- Em contrapartida, o VDM++ é uma linguagem de especificação formal orientada por objectos 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 mais importante no futuro, VDM++ prova o conceito agora

Exemplo: Modelo visual em UML e documentação associada

FStack
- elems: seq of int - capacity: nat
+ FStack(c: nat) + Push(x: int) + Pop() + Top(): int

- Atributos
  - elems – sequência que guarda os elementos guardados na stack, com o último inserido (topo da stack) à cabeça
  - capacity – número máximo de elementos
- Restrições:
  - O número de elementos guardados na stack não pode exceder a sua capacidade
- Operações
  - FStack – construtor, recebe como argumento a capacidade pretendida
  - Push – coloca um valor no topo da stack
  - Pop – remove o elemento que se encontra no topo da *stack* (último elemento inserido com Push), sem o retornar
  - Top – obtém o valor que se encontra no topo da *stack* (último elemento inserido com Push), sem o remover

Exemplo: Modelo formal e executável em VDM++

```
class FStack

 instance variables
   private elements : seq of int := [];
   private capacity:  nat;

  inv len elements <= capacity;

operations
  public FStack(c : nat) res: FStack ==
    (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 FStack
```

Valor antigo

Pré e pós-condições especificam semântica, corpo especifica algoritmo

- sqrt(x: real) : real
- Pré-condições - restrições nos dados de entrada e estado inicial:
  - x >= 0
- Pós-condições - efeito/resultado/estado final pretendido:
  - result * result = x
  - result >= 0
- Corpo - algoritmo
  - Linguagens como VDM++ permitem especificar o algoritmo a um nível de abstracção elevado
  - Corpo é necessário para executar / calcular resultado

Linguagens de especificação formal usam conceitos matemáticos

Tipo de dados composto	VDM++	OCL	Exemplo em VDM++
Conjunto de elementos do tipo *A*	set of A	Set(A)	{1 , 2}
Conjunto admitindo repetidos		Bag(A)	
Sequência de elementos do tipo *A*	seq of A	Sequence(A)	[1, 2]
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		{ 0 |-> false, 1 |-> true }
Mapeamento injectivo	inmap A to B		
Tuplo do 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, ...)	mk_(0, false) mk_T(0, false)
Conjunto de tuplos do tipo T (relação)	set of T	Set(T)	
União (alternativa)	A | B | ...		

Linguagens de especificação formal usam operadores matemáticos

Símbolo	ASCII
$\cdot$	&
$\times$	*
$\leq$	<=
$\geq$	>=
$\neq$	<>
$\xrightarrow{o}$	==>
$\rightarrow$	->
$\Rightarrow$	=>
$\Leftrightarrow$	<=>

Símbolo	ASCII
$\mapsto$	|->
$\triangleq$	==
$\uparrow$	**
$\dagger$	++
$\uplus$ (m)	munion
$\triangleleft$	<:
$\triangleright$	:>
$⩤$	<-:
$⩥$	:->
$\subset$	psubset
$\subseteq$	subset
$\frown$	^
$\bigcap$	dinter
$\bigcup$	dunion
$\mathcal{F}$	power
...-set	set of ...
$\ldots^{*}$	seq of ...
$\ldots^{+}$	seq1 of ...
$\ldots \xrightarrow{m} \ldots$	map ... to ...

Símbolo	ASCII
$\ldots \overset{m}{\longleftrightarrow} \ldots$	inmap ... to ...
μ	mu
$\mathbb{B}$	bool
$\mathbb{N}$	nat
$\mathbb{Z}$	int
$\mathbb{R}$	real
$\neg$	not
$\cap$	inter
$\cup$	union
$\in$	in set
$\notin$	not in set
$\wedge$	and
$\vee$	or
$\forall$	forall
$\exists$	exists
$\exists!$	exists1
λ	lambda
ι	iota
$\ldots^{-1}$	inverse ...

Quantificadores

Passos preconizados (1/2)

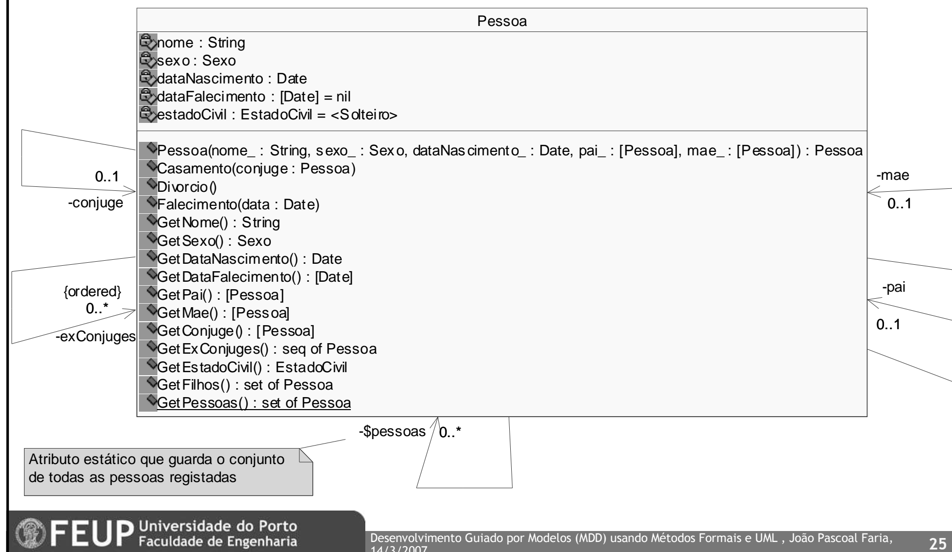
- Identificar requisitos
- Construir modelo visual (UML)
  - Modelo de casos de utilização
    - Incluindo possivelmente cenários de utilização/teste
  - Modelo de domínio
    - Incluindo possivelmente modelos de ciclos de vida de objectos
    - Incluindo possivelmente operações mais importantes (transacções e consultas)
  - Modelo captura requisitos e, inevitavelmente, alguma opções de desenho, mas inicialmente não há distinção de camadas!

Passos preconizados (2/2)

- Formalizar o modelo (VDM++)
  - Converter para VDM++
  - Definir tipos de dados (tipos de valores de atributos) em falta
  - Formalizar invariantes, pré-condições e (algumas) pós-condições
- Tornar o modelo executável (VDM++)
  - Escrever corpos de operações
- Verificar e validar o modelo
  - Escrever casos de teste, correr e analisar cobertura dos testes
  - Construir interface e ligar ao modelo para validação interactiva
  - Verificar consistência interna do modelo
- Refinar o modelo, gerar código, refinar o código, verificar e validar o código

Exemplo: Registo Civil

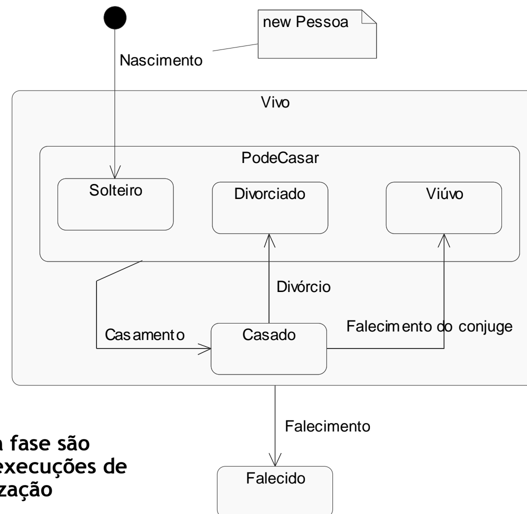
O diagrama de classes não basta …



... é necessário identificar restrições de integridade adicionais

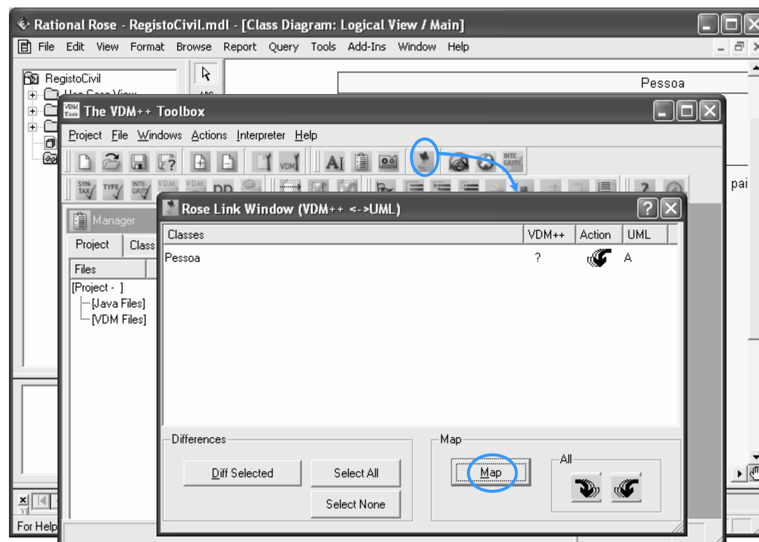
- **R1 -** O pai tem de ser do sexo masculino
- **R2 -** A mãe tem de ser do sexo feminino
- **R3** – Os cônjuges têm de ser de sexos opostos
- **R4 -** O falecimento tem de ser posterior ao nascimento
- **R5** – Os pais têm de nascer antes dos filhos

Diagrama de estados ajuda a especificar semântica de operações ...



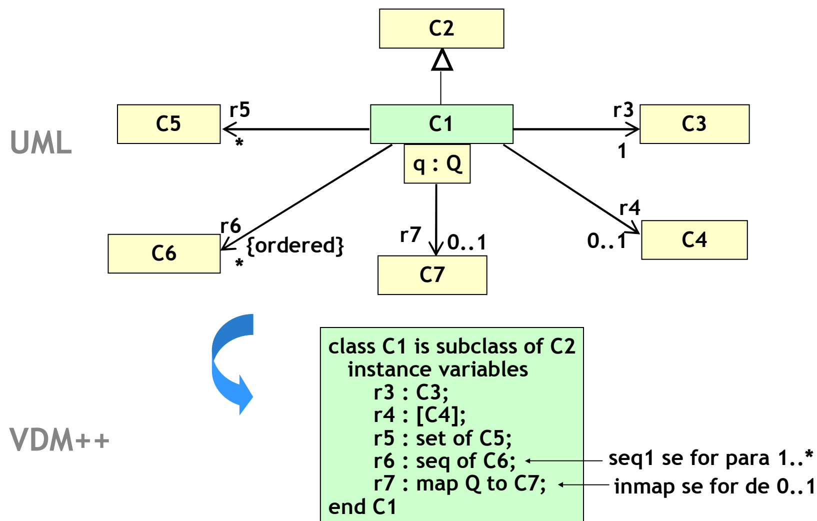
Eventos nesta fase são tipicamente execuções de casos de utilização

Importação de UML para VDM++



Regras de conversão

UML



VDM++

seq1 se for para 1..*

inmap se for de 0..1

Resultado da importação para VDM++ (1/2)

Mapeamento de atributos

Mapeamento de associações

```
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;
...
```

Ficheiro “Pessoa.rtf”

Resultado da importação para VDM++ (2/2)

Construtor

Transacções

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 GetEstadoCivil() res: EstadoCivil == is not yet specified;
  public GetFilhos() res: set of Pessoa == is not yet specified;
  public static GetPessoas () res: set of Pessoa == is not yet specified;

end Pessoa
```


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
```

Formalizar invariantes (1/3)

Função auxiliar → IsAfter

```
class Pessoa
 …
instance variables
  …

  -- O pai tem de ser do sexo masculino (R1) e ter data de
  -- nascimento anterior (R5)
  inv pai <> nil =>
          pai.sexo = <Masculino> and
          IsAfter(dataNascimento, pai.dataNascimento);

  -- A mãe tem de ser do sexo feminino (R2) e ter data de
  -- nascimento anterior (R5)
  inv  mae <> nil =>
           mae.sexo = <Feminino> and
           IsAfter(dataNascimento, mae.dataNascimento);
…
```


Formalizar invariantes (2/3)

```
…
  -- O falecimento não pode ser anterior ao nascimento (R4)
  inv dataFalecimento <> nil =>
          not IsAfter(dataNascimento, dataFalecimento);

  -- Os (ex)cônjuges têm de ser de sexos opostos (R3)
  inv conjuge <> nil => self.sexo <> conjuge.sexo;
  inv forall ex in set elems exConjuges & self.sexo <> ex.sexo;

  -- Simetria de "conjuge" e "exConjuges"
  -- (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;

…
```

Formalizar invariantes (3/3)

```
…
-- Consistência do estado civil (derivável parcialmente a partir doutros dados)
inv cases estadoCivil:
    <Solteiro>    -> conjuge = nil and exConjuges = [] and dataFalecimento = nil,
    <Casado>      -> conjuge <> nil and dataFalecimento = nil
                       and conjuge.dataFalecimento = nil ,
    <Divorciado> -> conjuge = nil and exConjuges <> [] and dataFalecimento = nil,
    <Viuvo>       -> conjuge = nil and exConjuges <> [] and dataFalecimento = nil
                       and exConjuges(len exConjuges).dataFalecimento <> nil,
    <Falecido>    -> conjuge = nil and dataFalecimento <> nil
   end;

operations
…
end Pessoa
```

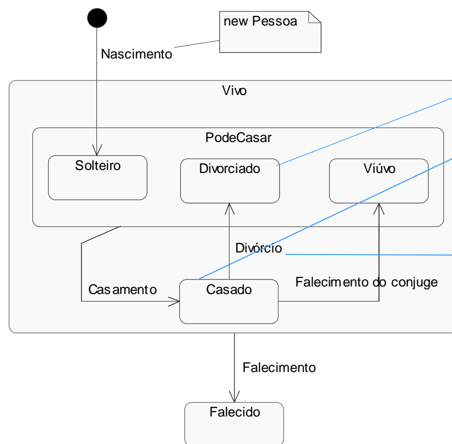

Invariantes orientados a conjuntos

- Se quisermos impor que não podem existir duas pessoas com o mesmo nome:

```
...
 -- Uniqueness constraint
 inv forall p, q in set pessoas &
        p <> q  => p.nome <> q.nome;
...
```

Quantificador universal

Formalizar pré/pós-condições: método



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

estadoCivil = <Divorciado>

estadoCivil = <Casado>

2º) Obter trivialmente :

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

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

Formalizar pré/pós-condições (1/4)

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 e em que modo

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

Mesmo que invariantes de estado, mas aplicados aos argumentos

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

Inicializa variáveis de instância com argumentos recebidos

post nome = nome0 and
  sexo = sexo0 and
  dataNascimento = dataNascimento0 and
  pai = pai0 and
  mae = mae0 and
  estadoCivil = <Solteiro> and

Acrescenta ao conjunto de instâncias

  pessoas = pessoas~ union {self} and

Retorna o próprio objecto

  res = self;

Formalizar pré/pós-condições (2/4)

```
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;
```

Actualiza o estado deste objecto (estadoCivil = <Casado>, conjuge = conj)

Actualiza o estado do outro objecto (conjuge) (conj.estadoCivil = <Casado>, conj.conjuge = self)

A operação é chamada para uma das pessoas do casal, e trata de actualizar o estado das duas pessoas.

Formalizar pré/pós-condições (3/4)

```
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
        conjuge~.exConjuges = conjuge~.exConjuges~ ^ [self];
```

Actualiza este objecto (estadoCivil, conjuge, exConjuges)

Actualiza o outro objecto (conjuge) (conjuge~.estadoCivil, conjuge~.conjuge, conjuge~.exConjuges)

A operação é chamada para uma das pessoas do casal, e trata de actualizar o estado das duas pessoas.

Formalizar pré/pós-condições (4/4)

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


Obter modelo executável

Corpo

```
public Pessoa(nome0: String, sexo0: Sexo, dataNascimento0: Date,
 pai0, mae0: [Pessoa]) res: Pessoa ==
(
  atomic(
    nome := nome0;
    sexo := sexo0;
    dataNascimento := dataNascimento0;
    pai := pai0;
    mae := mae0;
    pessoas := pessoas union {self};
  );
  return self
)
pre ...;
```

Corpo

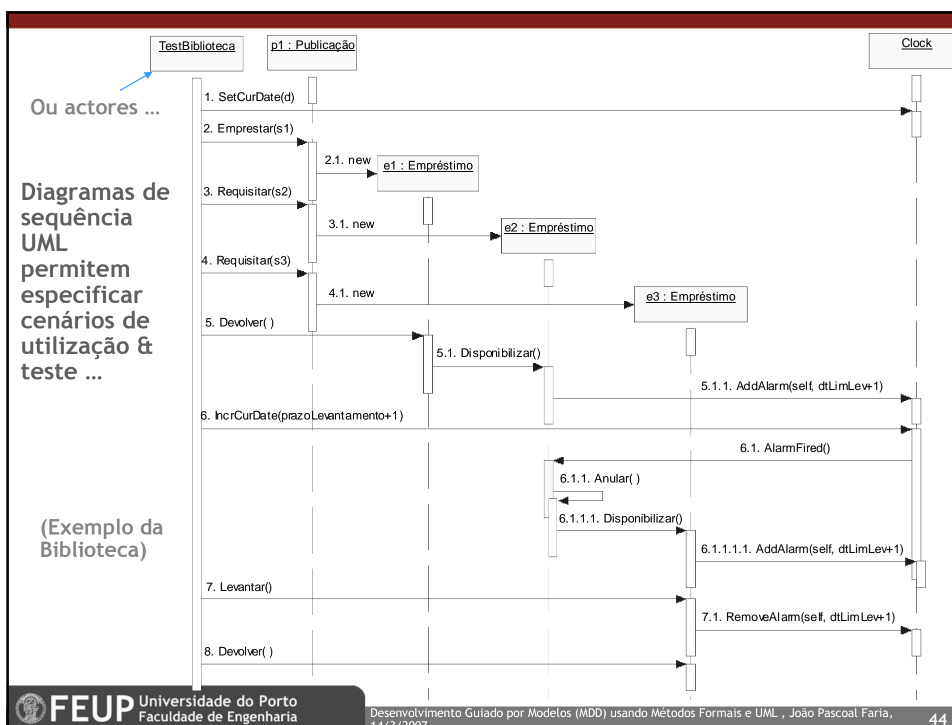
```
public GetFilhos() res : set of Pessoa ==
  return {f | f in set pessoas & f.GetPai()=self or f.GetMae()=self } ;
```

Definição de conjunto em compreensão

Etc.

Testar o modelo

- Modelo pode ser testado interactivamente (com interpretador de VDM++) ou com base em casos de teste definidos tb. em VDM++
  - Casos de teste podem ser gerados (semi-automaticamente) a partir de cenários de utilização descritos por diagramas de sequência em UML – ver a seguir
- Pode-se activar/desactivar a verificação automática de invariantes, pré-condições e pós-condições
  - Podem ser vistos como testes *built-in*
- É produzida informação dos testes que sucederam e dos testes que falharam
- É produzida informação de cobertura dos testes
  - o *pretty printer* "pinta" as partes da especificação que foram de facto executadas
  - são geradas tabelas com percentagem de cobertura e número de chamadas de cada operação e função

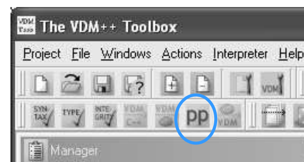


… convertíveis para casos de teste em VDM++

```
public TestScenario1(d: Date, p1: Publicação, s1, s2, s3: Sócio) ==
(
  dcl e1: Empréstimo;
  dcl e2: Empréstimo;
  dcl e3: Empréstimo;
  Clock`SetCurDate(d);
  e1 := p1.Emprestar(s1);
  e2 := p1.Requisitar(s2);
  e3 := p1.Requisitar(s3);
  e1.Devolver();
  Assert(e2.GetEstado() = <EsperaLevantamento>);
  Assert(e3.GetEstado() = <EsperaDisponibilidade>);
  Clock`IncrCurDate(Empréstimo`GetPrazoLevantamento()+1);
  Assert(e2.GetEstado() = <Anulado>);
  Assert(e3.GetEstado() = <EsperaLevantamento>);
  e3.Levantar();
  Clock`IncrCurDate(Empréstimo`GetPrazoDevolução()+1);
  Assert(s3.TemDevoluçõesAtrasadas());
  e3.Devolver()
);
```

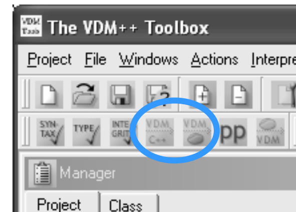

Documentação

- Cada classe VDM++ é especificada num ficheiro em classe.rtf, usando estilos pré-definidos para assinalar as partes de VDM++ (conforme conceito de programação literária)
- *Pretty printer* gera documento classe.rtf.rtf com formatação melhorada e informação de cobertura de testes
- Pode-se juntar tudo num documento final com "paste link"



Geração de código

- VDM Tools geram código completamente funcional em Java e C++
  - Facilmente legível no caso de Java
  - Geralmente pouco eficiente tanto em Java como em C++
- Maior dificuldade: utilizar bibliotecas pré-existentes da linguagem alvo desde a fase de especificação



Conclusões

- A integração de UML com uma linguagem como VDM++ permite enriquecer os modelos visuais criados em UML com:
  - especificação formal de restrições de estado e semântica de operações
  - especificação do corpo algorítmico de operações a um nível abstracto
- Obtém-se rapidamente uma especificação completa (sintaxe + semântica) e rigorosa do sistema a desenvolver a um nível de abstracção elevado
- Obtém-se rapidamente um modelo que pode ser executado, testado e validado
- Suporta-se a geração automática de código completamente funcional
- Facilita-se a verificação de conformidade da implementação com a especificação
- Ganha-se em qualidade e produtividade

Referências

- Sobre MDA:
  - "MDA Explained: The Model Driven Architecture -- Practice and Promise", by Anneke Kleppe, Jos Warmer and Wim Bast, from Addison-Wesley, 2003
  - www.omg.org
- Sobre UML:
  - www.uml.org – especificações e recursos sobre UML, OCL, etc.
- Sobre VDM++:
  - 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/