Métodos Formais em Engenharia de Software

Ana Paiva

apaiva@fe.up.pt

Universidade do Porto

Faculdade de Engenharia

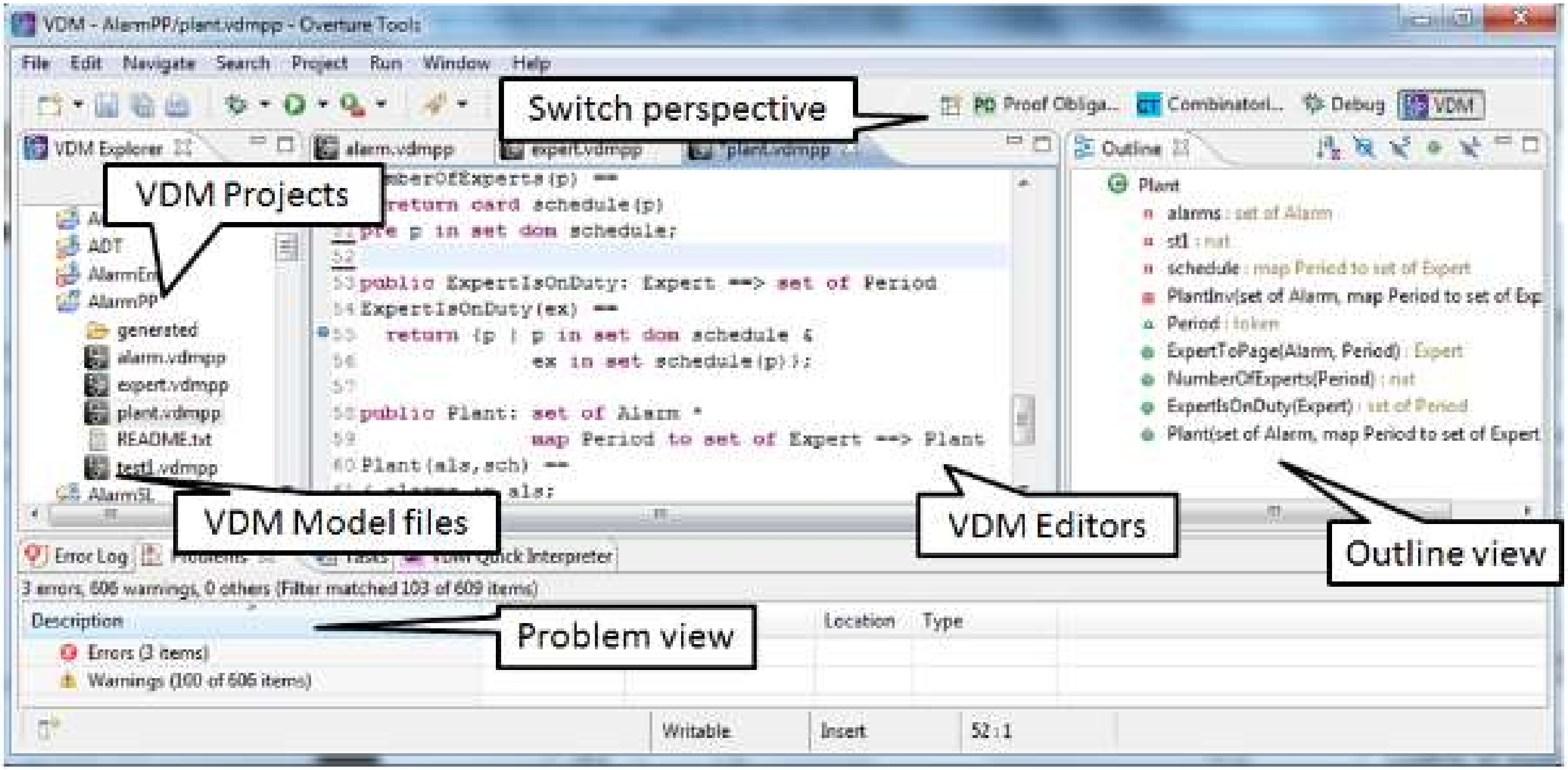
FEUP

Agenda

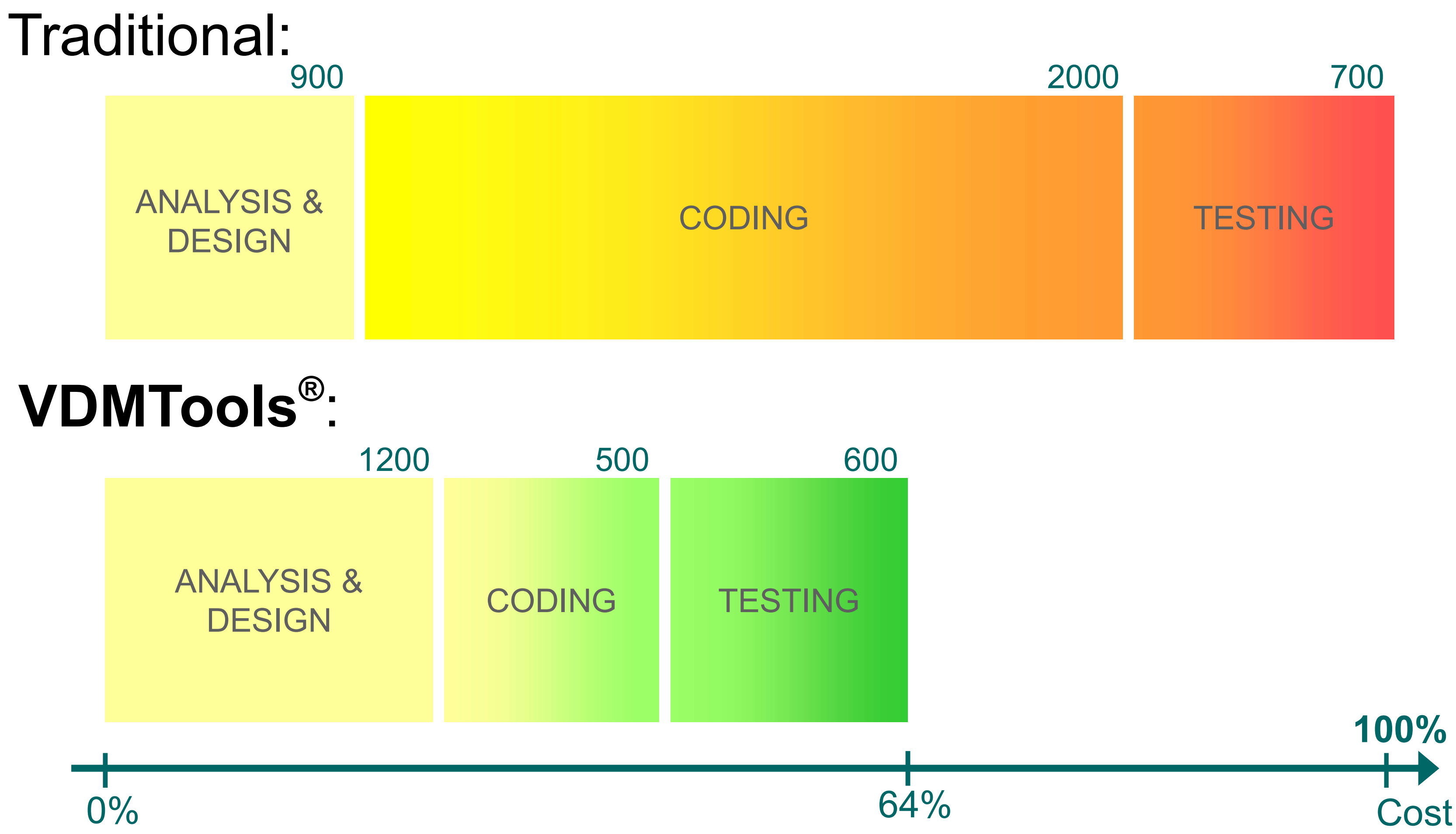
- VDMTools
- Characteristics of the VDM++ language
  - Classes; Instance variables; Operations; Functions (polymorphic, Higher-order functions, lambda, …); Types; Operators; Expressions
  - Design-by-contact:
    - Definitions of invariants; pre and postconditions
    - Link between VDM++ and UML
- Internal consistency: proof obligations
- Example: Vending Machine
- Concurrency in VDM++

Overture project

- http://www.overturetool.org/



Process



Traditional:
900
2000
700
ANALYSIS & DESIGN
CODING
TESTING
VDMTools®:
1200
500
600
ANALYSIS & DESIGN
CODING
TESTING
100%
0%
64%
Cost

Agenda

- VDMTools
- Characteristics of the VDM++ language
  - Classes; Instance variables; Operations; Functions (polymorphic, Higher-order functions, lambda, …); Types; Operators; Expressions
  - Design-by-contact:
    - Definitions of invariants; pre and postconditions
    - Link between VDM++ and UML
- Internal consistency: proof obligations
- Example: Vending Machine
- Concurrency in VDM++

Main characteristics of the VDM++

- Based on the standard VDM-SL (Vienna Development Method)
- Formal model based specification language (i.e., explicit representation of the state) object oriented
- Combination of two paradigm
  - Paradigm **functional**: types, functions and values
  - Paradigm **OO**: classes, instance variables, operations and objects
- Suported by VDMTools that allow:
  - The execution of an VDM++ specification
  - Specification testing and test coverage analysis
  - Synchronize with UML class diagrams in Rational Rose
  - Code generation to Java and C++
- Two notations available: ASCII and math symbols

VDM++ Class Outline

```
class <class-name>

  instance variables
    ...

  types
  values
  functions
  operations
    ...

  thread
    ...

  sync
    ...

end <class-name>
```

OO Paradigm	instance variables	Internal object state
Functional paradigm	types, values, functions	Definitions
OO Paradigm	operations	Definitions
	thread	Dynamic behaviour
	sync	Synchronization control

Classes

- A specification written in VDM++ is organized into classes
- Classes are reference types
  - Like what happens in several OO languages
  - Instances are mutable objects accessible by a reference
  - Variable of type C, where C is a class, contains a reference to the object with the data and not the data itself
  - Comparison and assignment operate with **references**
- Use to model the system state
  - State is represented by the set of existing objects and values of its instance variables
  - Classes represent types of physical entities (person, book room, ...), roles (teacher, student, ...), events (class, ...), documents (invoice, contract, ..., etc.).

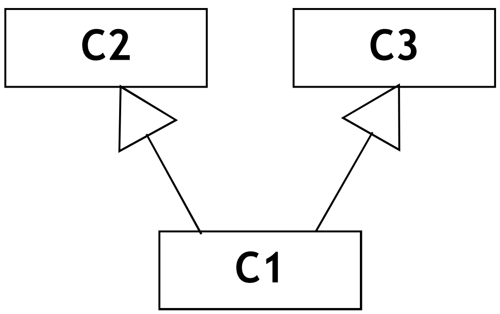
Inheritance

- A class may have several super-classes (multiple inheritance)
- Sintax:

    class C1 is subclass of C2, C3

    ...

    end C1



- Usual semantics
- Polymorphism

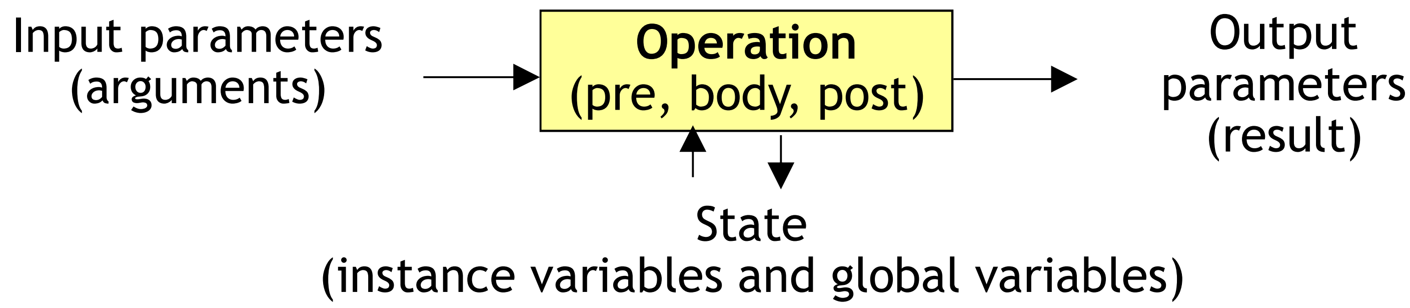
Instance variables

- Correspond to attributes in UML, and fields in Java and C#
- Can be private (by default), public or protected
- Can be static
- Declared in section “instance variables” with sintax:

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

- Can define invariants (*inv*) that restrict the set of valid values for the instance variables

Operations



- Correspond to operations in UML and methods in Java or C#
- Can be private (by default), public or protected
- They can be static
- Can view or modify the state of objects (given by instance vars) or the overall state of the system (given by static vars)
- May have pre-condition, body (explicit definition, mandatory) and post-condition (implicit definition)

Operations - definition

argument type result type

- Style 1:

```
op(a: A, b: B, …, z: Z) r: R ==
   bodystmt
```

omit when returns nothing

Variables read/written

```
ext rd instvarx, instvary, …
    wr instvarz, instvarw, …
pre preexpr(a, b, …, instvar1, instvar2, …)
post postexpr(a, b, …, r, instvar1, instvar2, …,
                   instvar1~, instvar2~, …) ;
```

- Style 2:

when there aren't neither arguments nor results, write ()

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

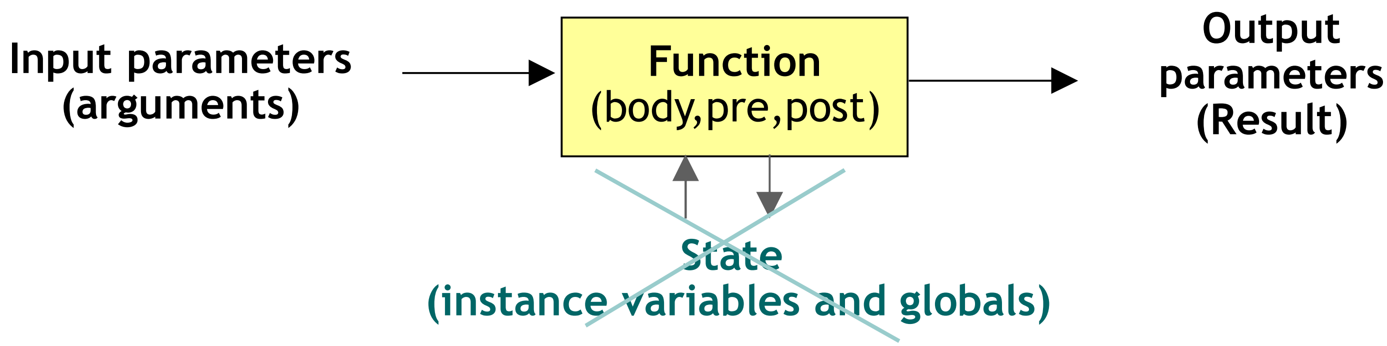
Predefined name for the return value

Variable state before the execution of the operation

Operation - definition

- Precondition (pre) - restriction on argument values and instance variables, to check the call
  - Can be omitted (even if true)
- Algorithmic body (bodysmt) – statement(s) between ()
  - Algorithm allows to express and execute the operation (explicit definition)
  - Imperative paradigm: c / assignments, variable declaration, etc..
  - Abstract operation "is subclass responsibility"
  - Operation to define: "is not yet specified," or omit "bodystmt ==" style 2.
- Postcondition (post) - the restriction on the values of the arguments, result, baseline and final vars (“~”, *Tilde*) instance, to check on return
  - Check the result / effect of the transaction (implicit definition)
  - Can be omitted (even if true)
- Clause "ext" (externals) - lists the instance variables that can be read (rd) and updated (wr) in the body of the operation
  - Required to indicate the style 1, when shown the postcondition

Functions - definition



- Pure functions without side effects, convert inputs into outputs
- Not have access (either read or change) the state of the system represented by instance variables
- Are defined in section *functions*
- They can be *private* (default), *public* or *protected*
- They can be *static* (normal case)
- May have pre-condition, body (for explicit definition, functional paradigm) and post-condition (for implicit definition)

Functions - definition

- Style 1:

May have several output parameters

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

- Style 2:

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

(simple or tuple)

Functions

- Body - explicit definition of the result(s) of the function by an expression without side effects
  - Functional paradigm, executable (to calculate the result)
  - We omit it: write "is not specified yet" or omit "bodyexpr ==" style 1
- Precondition (pre) - restriction on the values of the arguments that must check in function call
  - Allows you to define partial functions (not defined for some values of the arguments)
  - Can be omitted (even if true)
  - The precondition of a function f is also a function called pre_f
- Postcondition (post) - Boolean expression that relates the function result w / arguments (the restriction that it must obey the result)
  - Implicit definition of function (lets you check but not to calculate the result)
  - Can be omitted (even if true)
  - The post-condition of a function f is also a function called post_f

Functions – examples

- Given an implicit function, for example:

  ImplFn(n,m: **nat**, b: **bool**) r: **nat**
  pre n < m
  post if b **then** n = r **else** r = m

- There are two additional functions, automatically created, which can be used in the specification:

  pre_ImplFn: **nat** * **nat** * **bool** -> **bool**
  pre_ImplFn(n,m,b) ==
  n < m

  post_ImplFn: **nat** * **nat** * **bool** * **nat** -> **bool**
  post_ImplFn(n,m,b,r) ==
      if b
      then n = r
      else r = m

Functions – examples

- Explicit definition (executable), total function

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

- Implicit definition (not executable), partial function

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

- Recursive explicit function

```
fac: nat1 -> nat1
fac (n) ==
    if n > 1
    then n * fac(n-1)
    else 1
```

- Function with precondition

```
Division: real * real -> real
Division(p,q) ==
    p/q
pre q <> 0
```

Advanced functions in VDM++

- Polymorphic functions
- Higher-order functions
- The type function
- The lambda expression

Polymorphic functions

nomeFunção[@*TypeParam1*, @*TypeParam2*, ...] ...

- Are generic functions that can be used with different values
- They have special parameters (type parameters) that must be replaced by names of specific types using the function
- Names of these parameters start with "@" and are indicated in brackets after the function name
- Like function templates in C++

Polymorphic functions

- Example: utility function, which checks whether a sequence of elements of some kind has doubled:

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

- Example of its use:

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

A concrete type

The function type

- Total function: *arg1Type * arg2Type * … **+>** resultType*
- Partial function: $\underbrace{\textit{arg1Type * arg2Type * …}}_{}$ ***->*** *resultType*

It can be seen as a single package type argument tuple (instance calls of the product of types)

- The type of a Function is defined by the types of arguments and result
- Instances of a Function type (i.e., concrete function) can be passed as argument or as return, and saved (by reference) in data structures

Higher order functions

- Are functions that take other functions as arguments, or (Curried functions) that return functions as a result
- I.e. have arguments or a result of type function
- Example: a function that finds an approximate zero of a function between specified limits, with maximum error specified by the method of successive bisections:

```
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));
```

The function type

Operator	Name	Type
f1 comp f2	Function composition	(B->C) * (A->B) -> (A->C)
f ** n	Function iteration	(A->A) * nat -> (A->A)

Operator name	Semantics Description
Function composition	It yields the function equivalent to applying first f2 and then applying f1 to the result.
Function iteration	Yields the function equivalent to applying *f* *n* times. *n*=0 yields the identity function which just returns the values of its parameter; *n*=1 yields the function itself. For *n*>1, the result of *f* must be contained in its parameter type.

The lambda expression

- lambda *patternArg1*: *Type1*, …, *patternArgN*: *TypeN* & *expr*

λ

- Constructs a function on the fly
- Patterns are usually identifiers of arguments
- Normally used to pass as argument to another function (higher order)

Example: finding a real zero of a polynomial

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

Types

- Types are value types
  - Instances are immutable values pure
  - Comparison and assignment operate with their own values
  - Variable of type T name (a type) has its own data
- Subdivided into:
  - **Basic types** - bool, nat, real, char, ...
  - **Constructed types** (collections, etc..) - set of T, seq of T, map T1 to T2, ...
- New types can be defined within classes in the "types“ section
- The definition may include invariant for restricting valid instances
- Use to model types of values of attributes (data types)

Basic types

Symbol	Description	Examples of values
bool	Boolean	true, false
nat1	Natural number different from 0	1, 2, 3, …
nat	Natural number	0, 1, 2, …
int	Integer	…, -2, -1, 0, 1, …
rat **real**	Rational number Real number (the same as "rat" because only rational numbers can be represented in the computer)	-12.78, 0, 3, 16.23
char	Character	'a', 'b', '1', '2', '+', '-', …
token	Encapsulates a value (argument mk_token) of any type (useful if you know little about the type)	mk_token(1)
<identificador>	*Quotes* (literal names, typically used to define enumerated types)	<white>, <Black>

Constructed types - collections

Description	Sintax	Example of an instance
Set of elements of type A	set of A	{1 , 2}
Sequence of elements of type A	seq of A	[1, 2, 1]
Not empty sequence	seq1 of A	
Mapping elements of type A to type B elements (function finite set of key-value pairs)	map A to B	{ 0 |-> false, 1 |-> true }
Injected mapping (different key values correspond to different values)	inmap A to B	

More constructed types

Description	Sintax	Example of an instance
Products of types A, B, … (instances are tuples)	A * B * ...	mk_(0, false)
Record T with fields *a*, *b*, etc. of types *A*, *B*, etc. (*)	T :: a : A b : B ...	mk_T(0, false)
Union of types A, B, … (type A or type B or …)	A | B | ...	
Opcional type (allows **nil**)	[A]	

(*) Alternative definitions :

```
T :: a : A
     b :- B    -- field with ":-" is ignored in the comparison of records
```

Fields can be accessed by: **mk_T(x,y).b**

```
T :: A B     -- anonymous fields
```

Fields can be accessed by: **mk_T(x,y).#2**

Strings

- Not predefined type string, but can easily be defined as string (**seq of char**)
- All operations on sequences can be used with strings
- String literals can be indicated with quotation marks
  - “I am” is equivalent to ['I', ' ', 'a', 'm']

Example of a type definition

```
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
```

sequence

record

Enumerated type (defined with *union* and *quote*)

attribute

Data type

The type reference

- Reference to class object
- Allows the modeling of associations between classes and work with objects of classes
- Example: :

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

Guard reference to an object of class Person, or nil

Guard set of 0 or more references to objects of class Person

Symbolic constants

- Are constants which is given a name in order to make the specification more readable and easy to change
- Are declared in the section *values* with the syntax:

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

- Example:

```
values
    public PI = 3.14;
```

Boolean Operators

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**

Numeric operators

-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

Operators on sets (set)

Operador	Nome	Descrição	Tipo
e in set s1	In	$e \in s1$	A * set of $A \rightarrow$ bool
e not in set s1	Not in	$e \notin s1$	
s1 union s2	Union	$s1 \cup s2$	set of A * set of $A \rightarrow$ set of A
s1 inter s2	Intersection	$s1 \cap s2$	
s1 \ s2	Difference	$s1 \setminus s2$	
s1 subset s2	subset	$s1 \subseteq s2$	set of A * set of $A \rightarrow$ bool
s1 psubset s2	proper subset	$s1 \subset s2$ $(s1 \subseteq s2 \wedge s1 \neq s2)$	
s1 = s2	equal	$s1 = s2$	
s1 <> s2	Not equal	$s1 \neq s2$	
card s1	Cardinal	$\# s1$	set of $A \rightarrow$ nat
dunion ss	Distributed union	$\bigcup_{s_i \in ss} s_i$	set of set of $A \rightarrow$ set of A
dinter ss	Distributed intersection	$\bigcap_{s_i \in ss} s_i$	
power s1	Set of sets	$\mathscr{P}(s1)$	set of $A \rightarrow$ set of set of A

Exercises (sets)

- {1,…,6}
  - {1,2,3,4,5,6}
- {1,…,1}
  - {1}
- {4,…,1}
  - {}
- {x | x in set {2,3,4,5} & x > 2}
  - {3,4,5}
- {x | x in set {2,3,4,5} & 22 < x}
  - {}
- {} in set power {1,3,6}
  - true
- dunion {{1,2},{1,5,6},{3,4,6}}
  - {1,2,3,4,5,6}
- dinter {{1,2},{1,5,6},{3,4,6}}
  - {}
- {1,2,3} psubset {1,2}
  - false

Operators on sequences (seq)

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*

Exercises (seq)

- Which of the following expressions are true?
- 6 in set elems [3,6,8,10,0]
  - true
- [] = tl [4]
  - true
- 6 in set inds [3,6,8,10,0]
  - False

Exercises (seq)

- 2.2 What are the results of the following expressions:
- tl [1,2,3]
  - [2,3]
- len [[1,2],[1,2,3]]
  - 2
- hd [[1,2],[1,2,3]]
  - [1,2]
- tl [[1,2],[1,2,3]]
  - [[1,2,3]]
- elems [1,2,2,3,3,4]
  - {1,2,3,4}
- elems [[1,2],[2],[3],[3],[3,4]]
  - {[1,2],[2],[3],[3,4]}

Exercises (seq)

- 2.3 What is the value of the following expressions
- len []
  - 0
- len [1,2,3] + len [3]
  - 4
- [hd [<A>,<B>]] ^ [hd [<C>,<D>]]
  - [<A>,<C>]
- tl [1,2,3,4,5] ^ [hd [1,2,2]]
  - [2,3,4,5,1]
- tl ([1,2]^[1,2])
  - [2,1,2]

Operators on finite functions (maps)

Operador	Nome	Descrição	Tipo
dom m	Domain	Gives the domain (key set) of m	map *A* to *B* → set of *A*
rng m	Co-domain (range)	Gives the co-domain (set of values corresponding to keys) of m	map *A* to *B* → set of *B*
m1 munion m2	Merge	Makes a union of key-value pairs exist in m1 and m2, which must be compatible (they can not match different values to equal keys)	(map *A* to *B*) * (map *A* to *B*) → map *A* to *B*
m1 ++ m2	Override	Union with unrestricted compatibility. In case of dispute, m2 prevails.	
merge ms	Distributed union	Does the union of the mappings contained in ms that should be compatible.	set of (map *A* to *B*) → map *A* to *B*

Operators on finite functions (maps)

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`)	

Operators on finite functions (maps)

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*

Exercises (maps)

- dom {100 |-> <TIM>, 10 |-> <ROB>, 12 |-> <DAVE>}
  - {10,12,100}
- rng {100 |-> <TIM>, 10 |-> <ROB>, 12 |-> <DAVE>}
  - {<TIM>,<ROB>,<DAVE>}
- {1000 |-> 3, 1005 |-> 4, 1002 |-> 1} ++ {1002 |-> 6}
  - {1000 |-> 3, 1005 |->4, 1002 |->6}
- {1008 |-> 3, 1065 |-> 4, 1012 |-> 1} ++ {1011 |-> 6}
  - {1008 |-> 3, 1065 |-> 4, 1012 |-> 1, 1011 |-> 6}
- {128} <: {100 |-> <TIM>, 10 |-> <ROB>, 12 |-> <DAVE>}
  - {|->}
- {128} <-: {100 |-> <TIM>, 10 |-> <ROB>, 12 |-> <DAVE>}
  - {100 |-> <TIM>, 10 |-> <ROB>, 12 |-> <DAVE>}