
Fondamenti della Programmazione: Metodi Evoluti

Prof. Enrico Nardelli

Lezione 4: Contratti

Abstraction

To **abstract** is to capture the essence behind the details and the specifics.

The client is interested in:

- a **set of services** that a software module provides, not its internal **representation**

class

- **what** a service does, not **how** it does it

feature

- Object-oriented programming is all about finding right abstractions
- However, the abstractions we choose can sometimes fail, and we need to find new, more suitable ones.

Routine: algorithm abstraction

To abstract is to capture the *essence* of a concept, ignoring details & specifics

Implies:

- *Removing* some information
- Giving a *name* to the result of the abstraction

A routine is also known as a **method**, or a **subprogram** or a **subroutine**

In programming:

- Data abstraction: **class attributes**
- Algorithm (operational) abstraction: **class routine**

A routine is one of the two kinds of feature
... the other is *attribute*

A routine

r (*arg*: *TYPE*; ...)

-- Header comment.

require

Preconditions (boolean expression)

local

local variables

do

Body (instructions)

ensure

Postconditions (boolean expression)

end

Remember: two kinds of routine

Procedure: doesn't return a result

```
p (arg : TYPE; ...)  
  do  
    .....  
  end
```

- Yields a **command**
- Calls to a procedure are **instructions**

Function: returns a result

```
f (arg : TYPE; ...): RESULT_TYPE  
  ... (rest as before) ...
```

- Yields a **query**
- Calls to a function are **expressions**

Features: the full story

A class declaration is structured in clauses:

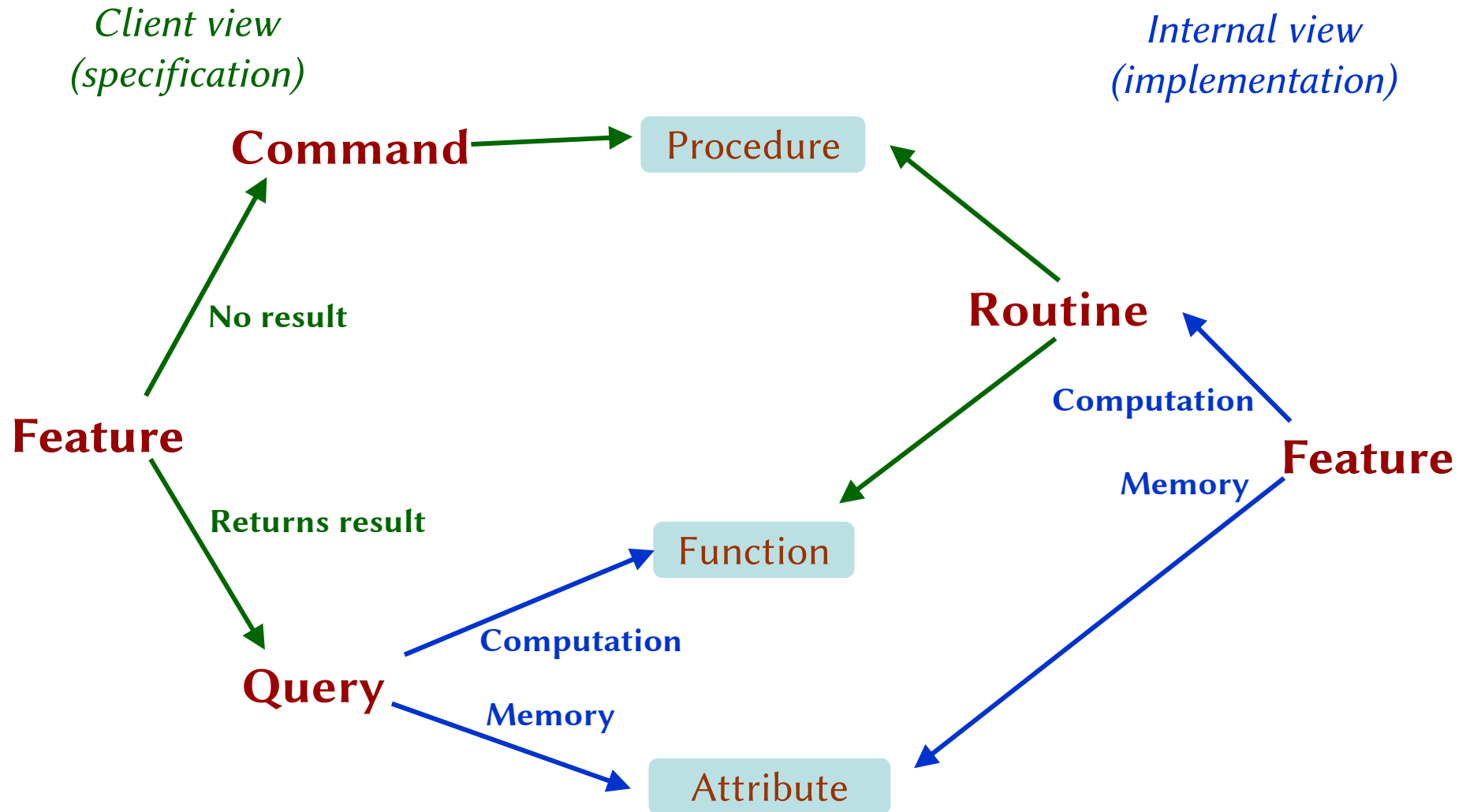
- Note
- Class
- Inherit
- Create
- Feature (any number of categories)
- Invariant
- End

A class is characterized by its features

Each feature is an operation on the corresponding objects:
query or command

Features are grouped into categories for readability (e.g. creation, access, status report, constants, basic operations, conversions, etc.)

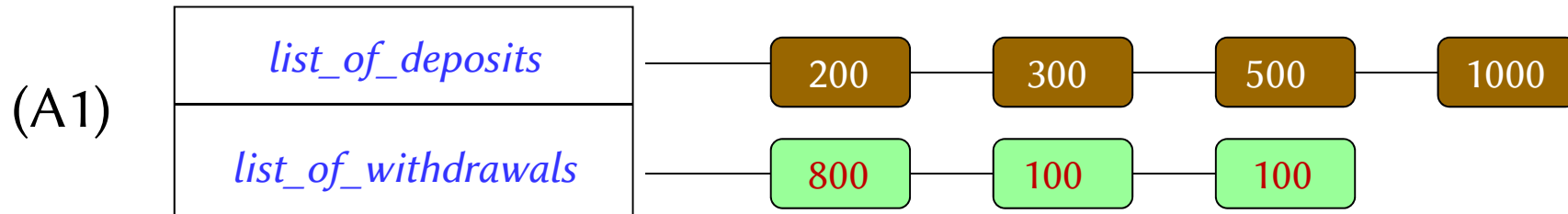
Features: the full story



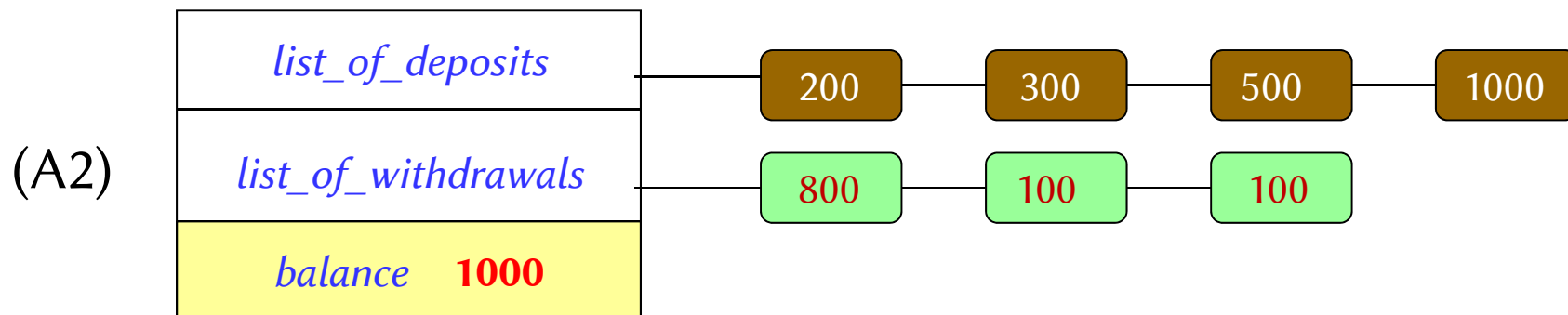
The Uniform Access principle

It doesn't matter to the client
whether you look up or compute

Uniform Access: an example



$balance = list_of_deposits.total - list_of_withdrawals.total$



A call such as

your_account.balance

could use an attribute or a function

The Uniform Access principle

Expressed more technically:

Features should be accessible to clients the same way
whether implemented by **storage** or by **computation**

Software construction

Finding appropriate classes is a central part of **software design**

Also called the development of the **architecture** of a program

Writing down the details is part of **implementation**

Style rule: header comments

Don't even *think* of writing a feature without immediately including a header comment explaining what it's about





Remember the **BANK_ACCOUNT** project?

Let's look at it again

First variation

We want to ensure only a positive sum is withdrawn

We want to ensure balance is always non negative

```
withdraw (sum: INTEGER)
```

```
-- Withdraw sum from the account
```

```
-- (Warning: use only if sum is positive and  $\geq$  balance)
```

Nice try, but...

...still not good enough:

- A comment is just an informal explanation
- The constraint needs a more official status in the interface

Contracts

A contract is a semantic condition characterizing correct usage properties of some construct

Three kinds for classes and features:

- Precondition
- Postcondition
- Class invariant

Specific contracts for iteration instructions:

- Loop invariant
- Loop variant

One generic version:

- Checking a property

Precondition

Property that a feature imposes on every client:

```
withdraw (sum: INTEGER)
```

```
-- Withdraw sum from the balance
```

```
require
```

```
not_negative: sum >= 0
```

```
covered: sum <= balance
```

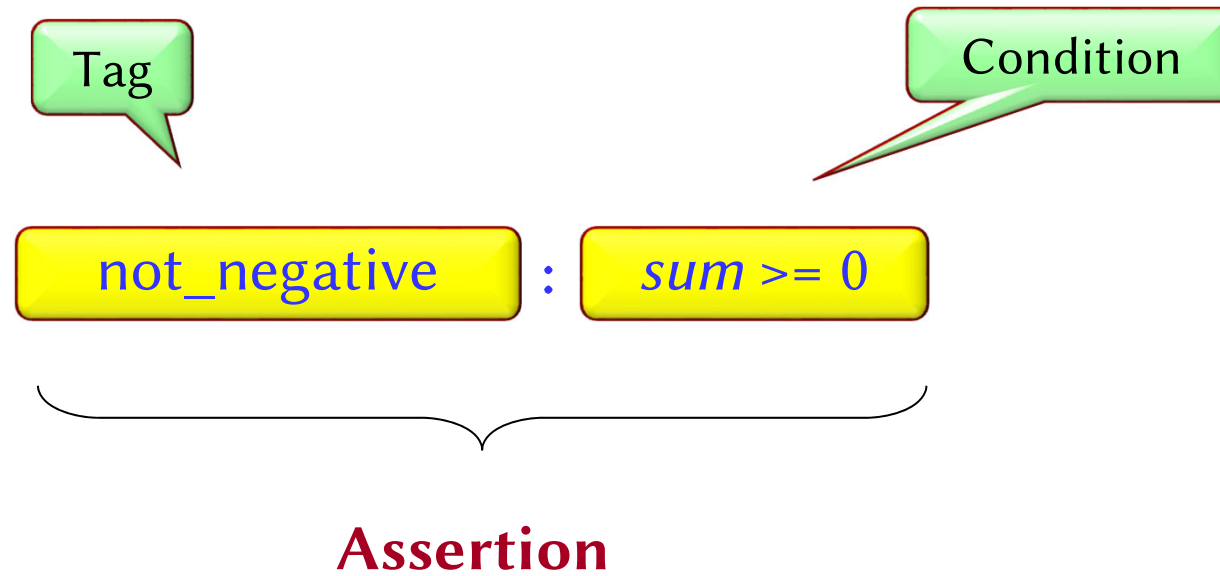
The precondition
of *withdraw*

A feature with no **require** clause is always applicable, as if it had

```
require
```

```
always_OK: True
```

Assertions



Precondition principle

A *client* calling a feature must make sure that the **precondition** holds before the call

A client that calls a feature without satisfying its precondition is faulty (**buggy**) software.

Postcondition

Property that a feature guarantees on termination:

close

-- Close the account balance

ensure

account_closed: *closed* = true

The postcondition
of *close*

A feature with no **ensure** clause always satisfies its postcondition, as if the postcondition reads

ensure

always_OK: **True**

Postcondition

Constraint on values **before** and **after** execution:

withdraw (sum: INTEGER)

-- Withdraw *sum* from the balance

ensure

decreased: *balance* = **old** *balance* - *sum*

Expression value
captured on entry

old notation

Denotes value of an expression as it was on routine entry

Usable **in postconditions only**

CANNOT be used in the body

Another example:

```
deposit (sum: INTEGER)
    -- Add sum to account.
    require
        positive: sum > 0
    do
        ...
    ensure
        added: balance = old balance + sum
    end
```

Postcondition principle

A *feature* must make sure that, if its precondition held at the beginning of its execution, its **postcondition** will hold at the end.

A feature that fails to ensure its postcondition is **buggy** software.

Preconditions and postconditions

Establish contractual relations between client and supplier

Precondition: obligation for clients

Postcondition: benefit for clients

All the clauses (assertions) in contracts must be true

They are checked in top down order

They are checked at run-time

Class invariants

The invariant expresses consistency requirements for instances of a class between feature calls

For a class REGULAR_ACCOUNT

invariant

limited: *balance* \leq *Max_amount*

Each clause of the class invariant must be true:

- before each feature execution
- after each feature execution

Comparison among contracts (1)

A **pre-condition** must be true before the execution of a feature, not necessarily afterwards.

A **post-condition** must be true after the execution of its feature, not necessarily before its execution or after the execution of other features

A **class invariant** must be true before/after the execution of **each** feature

Comparison among contracts (2)

A class invariant may be violated during the execution of code internal to a feature

Class invariants of x instance of C are **not** checked:

- when **leaving** the feature (before its termination) to execute
 - features of other objects
 - but class invariants of the called objects **are checked!**
 - other features of x (called through an unqualified call)
- when **re-entering** the feature after execution of other features

Contract to check a property

Use the **check** instructions (normally disabled in **finalized** mode)

Contains expression(s) ensuring that a certain property is satisfied at a specific point

Help document a piece of software

some_feature ...

do

... some implementation ...

check

tag_A : boolean_expression_stating_property_A

tag_B : boolean_expression_stating_property_B

...

end

... some implementation ...

end

Contracts

Contracts are useful for debugging: getting the software right

Contracts are useful for interface documentation, in particular, documenting API

Contracts execution is under compiler control (see Projects -> Settings under EiffelStudio)

Contract checking may be disabled in the finalized version for better performances

Contracts for iteration instructions will be seen later

Contracts outside of Eiffel

Java: Java Modeling Language (JML), iContract etc.

C#: Spec# (Microsoft Research extension)

UML: Object Constraint Language

Python

C++: Nana

etc.

**Let's add contracts to the
bank account example!**