Introduction to Eiffel

Martin Nordio
ETH Zurich
martin.nordio@inf.ethz.ch

Distributed and Outsourced Software Engineering - ETH course, Fall 2012

CONTRACTS

Contracts

A contract is a semantic condition characterizing usage properties of a class or a feature

Three principal kinds:

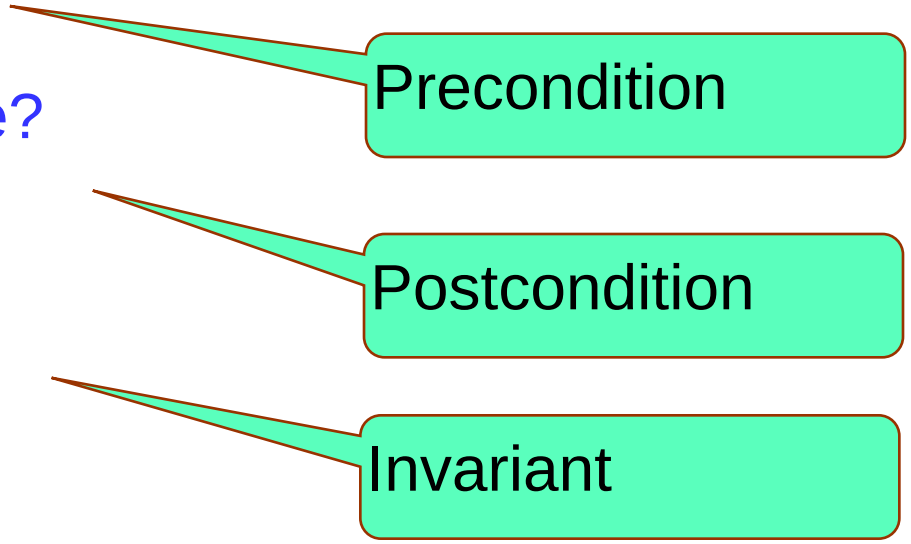
- Precondition
- Postcondition
- Class invariant

Design by Contract

Together with the implementation ("*how*") of each software element, describe "*what*" it is supposed to do: its contract

Three basic questions about every software element:

- What does it assume?
- What does it guarantee?
- What does it maintain?



Contracts in programming languages

Eiffel: integrated in the language

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

.Net languages: Code Contracts (a library)

Spec# (Microsoft Research extension of C#): integrated in the language

UML: Object Constraint Language

Python

C++: Nana

etc.

Precondition

Property that a feature imposes on every client:

```
factorial (i: INTEGER): INTEGER
    require
        valid_arg: i >= 0
    do
      ...
    end
```

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

```
require
    always_OK: True
```

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.

Precondition

Another example:

```
extend (a_element: G)
    require
        valid_elem: a_element /= void
        not_full: not is_full
    do  … end
```

A feature with a **require** clause

require

label_1: cond_1

label_2: cond_2 …

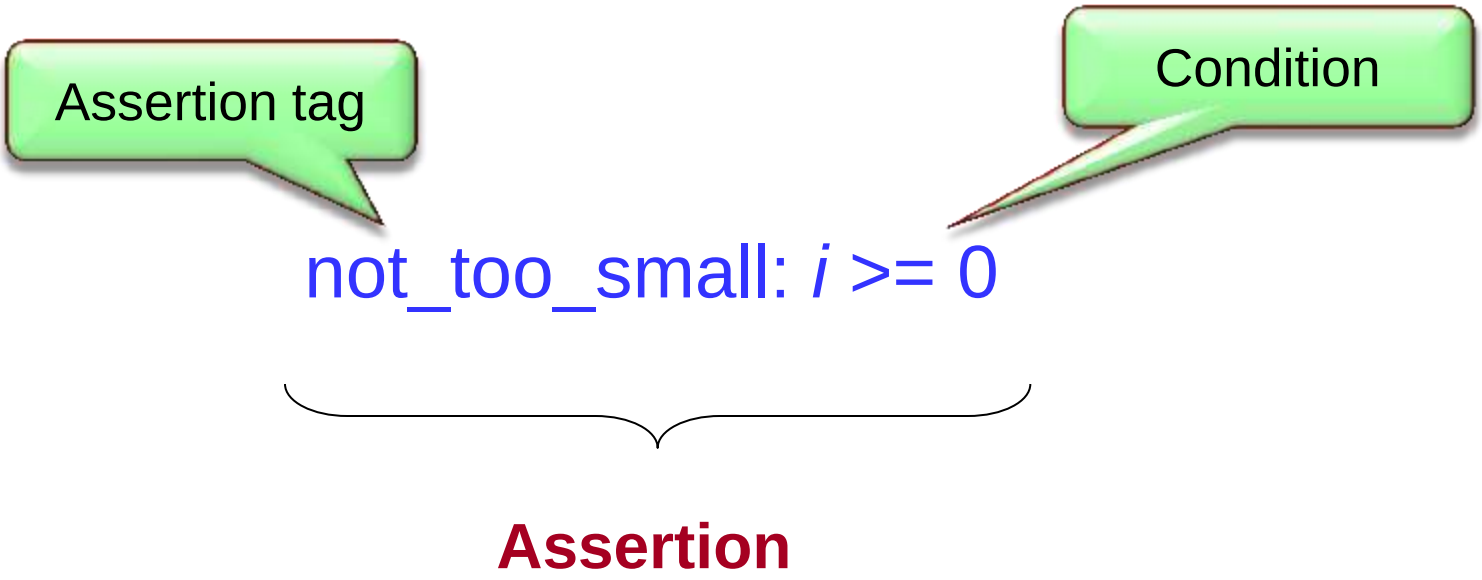
label_n: cond_n

is equivalent to

require

label: cond_1 **and** cond_2 **and …** *cond_n*

Assertions



Postconditions

Precondition: obligation for clients

Postcondition: benefit for clients

```
extend (a_element: G)
    ensure
        inserted: i_th (count) = a_element


index (a_element: G): INTEGER
    ensure
        exists: result > 0 implies i_th (result)  = a_element
        no_exists: result = -1 implies not is_inserted (a_element)
```

Old notation

Usable in postconditions only

Denotes value of an expression as it was on routine entry

Example (in a class ACCOUNT):

```
balance : INTEGER
        -- Current balance.

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

A class with contracts

```
class
    BANK_ACCOUNT
create
    make
feature
   make (n : STRING)
            -- Set up with name n
          require
              n /= Void
          do
                  name := n
                  balance := 0
          ensure
                  name = n
          end

   name : STRING
   balance : INTEGER
   deposit ( v : INTEGER)
            -- Add amount v
       do
          balance := balance + v
       ensure
              balance = old balance + v
       end
   invariant
       name /= Void
       balance >= 0
end
```


12

Contracts and inheritance

Issues: what happens, under inheritance:

Invariant Inheritance rule:

The invariant of a class automatically includes the invariant clauses from all its parents,

"and"-ed.

When redeclaring a routine, we may only:

Keep or weaken the precondition

Keep or strengthen the postcondition

Assertion redeclaration rule in Eiffel

A simple language rule does the trick!

Redefined version may have nothing (assertions kept by default), or

require else *new_pre*
ensure then *new_post*

Resulting assertions are:

- *original_precondition* **or** *new_pre*
- *original_postcondition* **and** *new_post*