Typology of programming languages

~ Contract Programming ~

What is that?

”It is absurd to make elaborate security checks on debugging runs, when no trust is put in the results, and then remove them in production runs, when an erroneous result could be expensive or disastrous. What would we think of a sailing enthusiast who wears his life-jacket when training on dry land but takes it off as soon as he goes to sea?”

–

Charles Antony Richard Hoare

Goals

In everyday life a service or a product typically comes with a contract or warranty: **an agreement in which one party promises to supply the service or product for the benefit of some other party**.

An effective contract for a service specifies requirements:

- Conditions that the consumer must meet in order for the service to be performed
  $\Rightarrow$ Preconditions
- Condition that the provider must meet in order for the service to be

Some History

- Has roots in work on formal verification, formal specification and Hoare logic
- First introducted by Eiffel
- Supported natively by Ada (2012), D, C#
- Librairies to emulate it in Java (cofoja), Javascript (contract.js), Python (pycontracts), C++ (Boost) ...

Contracts

A lot of ontracts:

- Pre-conditions and postconditions of a method
- Class invariants
- Assertions
- Loop invariants

Contracts

A lot of ontracts:

- Pre-conditions and postconditions of a method
- Class invariants
- Assertions
- Loop invariants

Contracts are part of the language:

- a dedicated syntaxe
- compiled (or not) according to the given options
- used by the compiler
- used by the environnemnt
- used by the documentation

Pre-conditions

Pre-conditions must be fulfill by the client, i.e. based on arguments

```
class SHAPE

feature
   xc, yc : INTEGER ; -- coordinates

   set_x_y(x,y : INTEGER) is
      require
         x >= 0 and y >= 0
      do
         xc = x ;
         yc = y ;
      end ;
...
```

Pre-conditions in **Eiffel**

Post-conditions

Post-conditions must be fulfill by the provider, i.e. if the client fulfill preconditions, the provider will fulfill postcondiitons.

```
class SHAPE

feature
   ...
   set_x_y(x,y : INTEGER) is
      require
         x >= 0 and y >= 0
      do
         xc := x ;
         yc := y ;
      ensure
         xc = x and yc = y
      end ;
```

Referencing previous version of an expression

`old x` reference the value of x before the execution of the method

```
class RECTANGLE

feature
   width, height : INTEGER ;

   set_width(w : INTEGER) is
      require
         w > 0
      do
         width := w
      ensure
         width = w and height = old height
      end ;
...
```

Referencing previous value in Post-

Stripping Objects

In a postcondition, `strip(x,y,..)` references an object where all attributes *x* and *y*, ... have been removed

```
class RECTANGLE
feature
 width, height : INTEGER ;

 set_width(w : INTEGER) is
   -- change the width
   require
      w > 0
   do
      width := w
   ensure
     width = w and
       strip (width) = old strip (width)
   end ;
```

Redefinition (1/2)

class A

```
routine p is require ... ensure ... end ;

routine q is do p() ; end ;
```

redefine p

class B

```
routine p is do ... end ;
```

The redefined method p in B can be used instead of the original method p de A.
⇒ Assertions are inherited

Redefinition (2/2)

The redefined method must satisfy old assertions but can be more precise:

- Release some preconditions
- Add (Restrict) postconditions

```
class B

inherit
   A redefine p end ;

feature
   p is
      require else
         ... -- other restrictions for calls
      do
         ... -- new defintion
      ensure then
         ... -- additionnal postconditions
      end ;
end -- class
```

Class Invariants

A Class Invariant is an assertion attached to an object. The inherited class also inherits invariants.

```
class RECTANGLE
   ...

   invariant
         (xc < 0 implies width > -xc)
      and
         (yc < 0 implies height > -yy)
      and
         width >= 0
      and
         height >= 0

end -- class RECTANGLE
```

Assertions

Can be inserted anywhere in the code.

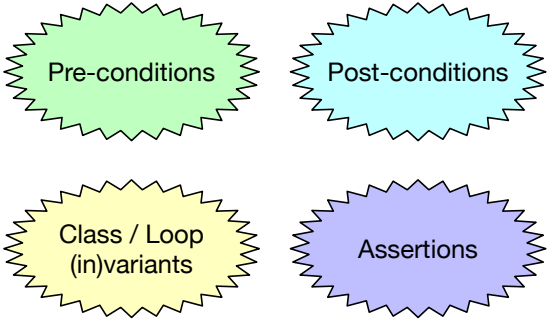
```
-- Code
check
   x > 0 ;
   y < 0 implies width > -y
end ;
```

Loop (in)variants

Only one (complex) kind of loop in Eiffel

```
from
   -- initialization
   ...
invariant
   -- checked each iteration
   ...
variant
   -- positive integer expression
   ...
until
   -- exit condition
   ...
loop
   -- loop body
   ...
end ;
```

Summary


Pre-conditions
Post-conditions
Class / Loop
(in)variants
Assertions