Lecture 14: More Inheritance & Genericity

Genericity: summary 1

Type extension mechanism
Reconciles flexibility with type safety
Enables us to have parameterized classes
Useful for container data structures: lists, arrays, trees, ...
"Type" now a bit more general than "class"

(Reminder) A generic class

class LIST[G] feature
    extend(x: G) is ...
    last: G is ...
end

To use the class: obtain a generic derivation, e.g.

cities: LIST[CITY]

(Reminder) Using generic derivations

cities: LIST[CITY]
people: LIST[PERSON]
c: CITY
p: PERSON
...
cities.extend(c)
people.extend(p)
c := cities.last
c.add_tram_line(Line8)

STATIC TYPING:
The compiler will reject:
  ➢ people.extend(c)
  ➢ cities.extend(p)

(Reminder) Using genericity

LIST [CITY]
LIST [LIST [CITY]]
...
A type is no longer exactly the same thing as a class!
(But every type remains based on a class.)
Genericity: summary 1

Type extension mechanism
Reconciles flexibility with type safety
Enables us to have parameterized classes
Useful for container data structures: lists, arrays, trees, ...
"Type" now a bit more general than "class"

Definition: Type

We use types to declare entities, as in

\[ x : \text{SOME\_TYPE} \]

With the mechanisms defined so far, a type is one of:

\[ \begin{align*}
& \text{A non-generic class} \\
& \quad \text{e.g. METRO\_STATION} \\
& \text{A generic derivation, i.e. the name of a class followed} \\
& \quad \text{by a list of types: the actual generic parameters, in} \\
& \quad \text{brackets} \\
& \quad \text{e.g. LIST[METRO\_STATION]} \\
& \quad \text{LIST[ARRAY[METRO\_STATION]]}
\end{align*} \]

Genericity + inheritance 1: Constrained genericity

```plaintext
class VECTOR [G] feature
  plus alias "+" (other VECTOR[G]): VECTOR[G] is
    -- Sum of current vector and other
    require
      lower <= other.lower
      upper >= other.upper
    local
      a, b, c: G
    do
      ... See next ...
    end
    ... Other features, in particular item to access an 
    element ...
end
```

Adding two vectors

\[ u + v = w \]

The solution

Declare class VECTOR as

```plaintext
class VECTOR [G -> NUMERIC] feature
  plus alias "+", minus alias "-"and so on.
end
```

Class NUMERIC (from the Kernel Library) provides features plus alias "+", minus alias "-"and so on.
Improving the solution

Make `VECTOR` itself a descendant of `NUMERIC`, effecting the corresponding features:

```plaintext
class VECTOR [G -> NUMERIC] inherit NUMERIC
  feature
    ... Rest as before, including infix "+"...
  end
```

Then it is possible to define:

```plaintext
v : VECTOR [INTEGER]
v : VECTOR [VECTOR [INTEGER]]
vv : VECTOR [VECTOR [VECTOR [INTEGER]]]
```

Extending the basic notion of class

```plaintext
Inheritance
  SET OF CARS
  LIST OF CARS
  LIST OF CITIES
  LIST OF PERSONS
  LIST OF FIGURE
  LINKED LIST OF CARS
  SET OF PERSONS

Genericity
```

Genericity + Inheritance 2: Polymorphic data structure

```plaintext
class LIST [G]
  feature
    last : G is ...
    extend(x : G) is ...
  end

fl : LIST [FIGURE]
r : RECTANGLE
s : SQUARE
```

```plaintext
fl.extend (p); fl.extend (t); fl.extend (s); fl.extend (r)
fl.last.display
```

Example hierarchy

```plaintext
OPEN FIGURE
  POLYLINE
    POLYGON
      CIRCLE
    TRIANGLE
      SQUARE
  ELLIPSE
  CLOSED FIGURE
```

Forcing a type: the problem

```plaintext
fl.store("FILE_NAME")
...
-- Two years later:
  fi := fl.last   -- [1]
  print(x.diagonal) -- [2]
```

What’s wrong with this?

> If `x` is declared of type `RECTANGLE`, [1] is invalid.
> If `x` is declared of type `FIGURE`, [2] is invalid.

The solution: Assignment attempt

```plaintext
fl.store("FILE_NAME")
...
-- Two years later:
  fi := fl.retrieve("FILE_NAME") - See next
  x := fl.last   -- [1]
  print(x.diagonal) -- [2]
```

But:

If `x` is declared of type `RECTANGLE`, [1] is invalid.
If `x` is declared of type `FIGURE`, [2] is invalid.
Assignment attempt

```plaintext
f: FIGURE
r: RECTANGLE
...
fl.retrieve("FILE_NAME")
f := fl.last
r := f
if r /= Void then
  print(r.diagonal)
else
  print("Too bad.")
end
```

Assignment attempt

```plaintext
x ?= y
with
  x : A
If y is attached to an object whose type conforms to A,
perform normal reference assignment.
Otherwise, make x void.
```

The role of deferred classes

Express abstract concepts independently of implementation
Express common elements of various implementations

Terminology: Effective = non-deferred
(i.e. fully implemented)

A deferred feature

In e.g. LIST:
```
forth
not
after
ensure
index = old index + 1
```

Mixing deferred and effective features

In the same class
```
(x: G)
-- Move to first position after current
-- where x appears, or after if none.
do
  from until after or else item = x loop
  forth
end
end
```

"Don’t call us, we’ll call you!"

A powerful form of reuse:
> The reusable element defines a general scheme
> Specific cases fill in the holes in that scheme

Combine reuse with adaptation
Applications of deferred classes

Analysis and design, top-down
Taxonomy
Capturing common behaviors

Deferred classes in EiffelBase

Java and .NET solution

Single inheritance only for classes
Multiple inheritance from interfaces
An interface is like a fully deferred class, with no implementations (do clauses), no attributes (and also no contracts)

Inheritance and assertions

Correct call:
if a1.a then
  a1.r(...)  
else
  ...
end

Assertion redeclaration rule

Redefined version may not have require or ensure.
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

Invariant accumulation

Every class inherits all the invariant clauses of its parents. These clauses are conceptually "and"-ed.
What we have seen

- Genericity
- Combining inheritance with genericity
- Constrained genericity
- Forcing a type: the assignment attempt
- The relationship of contracts with inheritance
- Deferred classes and features

End of lecture 14