Introduction to Programming
Bertrand Meyer

Lecture 13: Inheritance & genericity

Last revised 11 December 2005

Inheritance basics

Principle:
- describe a new class as extension or specialization of an existing class (or several with multiple inheritance)

If $B$ inherits from $A$:
- Module viewpoint: all the services of $A$ are available in $B$ (possibly with different implementation).
- Type viewpoint: whenever an instance of $A$ is required, an instance of $B$ will be acceptable ("is-a" relationship)

Terminology

Parent, Heir
Ancestor, Descendant
- The ancestors of $B$ are $B$ itself and the ancestors of its parents.
- Proper ancestor, Proper descendant

Direct instance, Instance
- The instances of $A$ are the direct instances of its descendants.

(Other terminology: subclass, superclass, base class)

Example hierarchy

Redefinition 1: polygons

class POLYGON inherit
CLOSED FIGURE
create
make
feature
vertex: ARRAY[POINT]
vertex_count: INTEGER
perimeter: REAL
-- Perimeter length
   do
   from ... until ...
   Result = Result + vertex[1].distance(vertex[/+ 1])
   end
invariant
vertex_count >= 3
vertex_count = vertex_count + vertex_count

Redefinition 2: rectangles

class RECTANGLE inherit
POLYGON
redefine
perimeter
create
make
feature
diagonal, side1, side2: REAL
perimeter: REAL
-- Perimeter length
   do Result = Result + side1 + side2
   end
invariant
vertex_count = 4
Inheritance, typing and polymorphism

- Assume:
  - \( p \): POLYGON, \( r \): RECTANGLE, \( t \): TRIANGLE,
  - \( x \): REAL

- Permitted:
  - \( x := p \).perimeter
  - \( x := r \).perimeter
  - \( x := r \).diagonal

- NOT permitted:
  - \( x := p \).diagonal (even just after \( p := r \))
  - \( r := p \)

Dynamic binding

- What is the effect of the following (assuming \texttt{some_test} true)?
  - if \texttt{some_test} then
  - \( p := r \)
  - else
  - \( p := t \)
  - end

- \( x := p \).perimeter

- Redefinition: A class may change an inherited feature, as with \texttt{POLYGON} redefining \texttt{perimeter}.
- Polymorphism: \( p \) may have different forms at run-time.
- Dynamic binding: Effect of \( p \).\texttt{perimeter} depends on run-time form of \( p \).

Without inheritance!

\[
\text{display}(f \texttt{.FIGURE}) \text{ do}
\]
\[
\text{if "f is a CIRCLE" then}
\]
\[
\text{else "f is a POLYGON" then}
\]
\[
\text{end}
\]
\[
\text{end}
\]

and similarly for all other routines!

Tedious, must be changed whenever there's a new figure type.

With inheritance!

With:

- \( f \texttt{.FIGURE} \)

and:

- \( c \texttt{.CIRCLE} \)

- \( p \texttt{.POLYGON} \)

Initialize:

- \( f := c \)
- \( f := p \)

Then just use:

- \( f \texttt{.move(...)} \)
- \( f \texttt{.rotate(...)} \)
- \( f \texttt{.display(...)} \)

-- and so on for every -- operation on \( f \)!

Inheritance: summary 1

- Type mechanism: lets you organize our data abstractions into taxonomies
- Module mechanism: lets you build new classes as extensions of existing ones
- Polymorphism: Flexibility with type safety
- Dynamic binding: automatic adaptation of operation to target, for more modular software architectures

Genericity

Unconstrained

\[
\text{LIST} [G]
\]
\[
\text{e.g. \text{LIST} [INTEGER], \text{LIST} [PERSON]}
\]

Constrained

- \( \text{HASH\_TABLE} [G \rightarrow \text{HASHABLE}] \)
- \( \text{VECTOR} [G \rightarrow \text{NUMERIC}] \)
**Extending the basic notion of class**

- Abstraction
- Inheritance
- Type parameterization
- Genericity

**Genericity: Ensuring type safety**

How can we define consistent "container" data structures, e.g. list of accounts, list of points?

Dubious use of a container data structure:

```plaintext
Dubious use of a container data structure:

c: CITY; p: PERSON

cities: LIST...
people: LIST...
people.extend (p)
cities.extend (c)

c := cities.last

c.add_tram_line (Line8)
```

**Possible approaches**

1. Duplicate code, manually or with help of macro processor.
2. Wait until run time; if types don’t match, trigger a run-time failure (Smalltalk)
3. Convert (“cast”) all values to a universal type, such as “pointer to void” in C.
4. Parameterize the class, giving an explicit name G to the type of container elements. This is the Eiffel approach, now being adopted also by Java, .NET and others.

**A generic class**

- Formal generic parameter

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

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

```plaintext
cities: LIST[CITY]
c := cities.last
c.add_tram_line (Line8)
```

**Using generic derivations**

```plaintext
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)`
### Static typing

**Type-safe call** (during execution):
- A feature call \( x.f \) such that the object attached to \( x \) has a feature corresponding to \( f \).
  - (generalizes to calls with arguments \( x.f(a, b) \))

**Static type checker**:
- A program-processing tool (such as a compiler) that guarantees, for any program it accepts, that any call in any execution will be type-safe.

**Statically typed language**:
- A programming language for which it is possible to write a static type checker.

### The static and the dynamic

For a feature call \( x.f \):
- **Static typing**:
  - There is at least one feature \( f \) applicable to \( x \)
- **Dynamic binding**:
  - If more than one possible feature, execution will select the right feature

### Using genericity

- \( \text{LIST [CITY]} \)
- \( \text{LIST [LIST [CITY]]} \)
- ...

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"

### 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 ...
end
```

### Adding two vectors

\[
U + V = W
\]

```
2
1
a + b = c
```

```plaintext
\[
\begin{array}{c}
2 \\
1 \\
a \\
b \\
c
\end{array} + \begin{array}{c}
2 \\
1 \\
a \\
b \\
c
\end{array} = \begin{array}{c}
2 \\
1 \\
c
\end{array}
\]```
Constrained genericity

Body of plus alias "•":

create Result.make (lower, upper)
from
i := lower
until
i => upper
loop
a := item(i)
b := other.item(i)
c := a * b     -- Requires "•" operation on G!
Result.put (c, i)
i := i + 1
end

The solution

Declare class VECTOR as

class VECTOR [G => NUMERIC] feature
    ... The rest as before ...
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:

class VECTOR [G => NUMERIC] inherit
    NUMERIC
    feature
        ... Rest as before, including infix "•" ...
    end

Then it is possible to define:

v: VECTOR [INTEGER]
vv: VECTOR [VECTOR [INTEGER]]
vvv: VECTOR [VECTOR [VECTOR [INTEGER]]]

Extending the basic notion of class

Genericity

List of CARS

List of MACHINES

List of PERSONS

Inheritance

Linked List of CARS

Linked List of MACHINES

Linked List of PERSONS

Example hierarchy

List of CARS

List of MACHINES

List of PERSONS

Open Figure

Closed Figure

Segment

Polyline

Polygon

Ellipse

Corner

Point

SQUARE

RECTANGLE

POLYGON

TRIANGLE

OPEN FIGURE

CLOSED FIGURE

SEGMENT

POLYLINE

POLYGON

ELLIPSE

Corner

Point

SQUARE

RECTANGLE

CIRCLE

POLYGON

TRIANGLE

Example hierarchy
Forcing a type: the problem

```java
fl.store("FILE_NAME")
...
-- Two years later:
fl := retrieved("FILE_NAME")
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.

The solution: Assignment attempt

```java
fl.store("FILE_NAME")
...
-- Two years later:
fl? := retrieved("FILE_NAME")
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

```java
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

```java
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.
```

Inheritance and assertions

```java
at: A
at_r(...)    r is require or ensure β
```

Correct call:

```java
if at.a then
    at_r(...)  r is require or ensure β
else
    ...
end
```

Assertion redeclaration rule

- Redefined version may not have require or ensure.
- May have nothing (assertions kept by default), or
  ```java
  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.

End of lecture 13