Introduction to Programming

Till G. Bay

Last revised 6 January 2004
Lecture 18:
Inheritance Revisited

- Inheritance
  - Terminology
  - Example
  - Polymorphism and dynamic binding
- Genericity
  - Assignment attempt
  - Constrained genericity
- Inheritance and contracts
What is inheritance?

- Describe a new class as extension or specialization of an existing class. (With MULTIPLE inheritance it can be an extension of several existing classes.)
What is inheritance?

class A
attribute a
  do
    -- Some Code
  end
end

class B
attribute b
  do
    -- Some Code
  end
end

class C
  inherit A
  feature
    c
      do
        -- Some Code
      end
  a
    do
      -- Some Code
    end
end

class D
  inherit A, B
  attribute c
    do
      -- Some Code
    end
  a
    do
      -- Some Code
    end
  b
    do
      -- Some Code
    end
end
- **Parent, Child**
- **(Other terminology: superclass, subclass)**
A deferred class has *at least one* deferred feature. A deferred feature is a feature that has no feature body but only the feature declaration with its signature.
A effective class is a class that inherits from a deferred class and implement at least one of the deferred features of the deferred class.
Terminology: Redefinition of a feature of a parent class

C and D are normal classes – they are not deferred and not effective. Here they are used to show how a feature $f$ from class C can be redefined in class D.
Example: Inheritance hierarchy

- FIGURE
  - extent*
  - center*
  - rotate*
  - display*
- OPEN FIGURE
  - perimeter*
- CLOSED FIGURE
  - perimeter*

- POLYGON
  - perimeter+
  - side1
  - side2

- RECTANGLE
  - perimeter++
  - diagonal

- ELLIPSE
  - perimeter+

- CIRCLE
  - perimeter++

* deferred
+ effective
++ redefined
Example: POLYGON

class POLYGON create
    make
feature
    vertices: ARRAY [POINT]
    vertices_count: INTEGER
    perimeter: REAL is
        -- Perimeter length
        do
            from ... until ... loop
                Result := Result +
                    (vertices @ i).distance (vertices @ (i + 1))
            end
        end

    invariant
        vertices_count >= 3
        vertices_count = vertices.count
end
Example: RECTANGLE by redefining POLYGON

class RECTANGLE inherit
  POLYGON
  redefine
  perimeter
  end
create
  make
feature
  diagonal, side1, side2: REAL
  perimeter: REAL is
    -- Perimeter length
    do
      Result := 2 * (side1 + side2)
    end
invariant
  vertices_count = 4
end
Assume:

\[ p: POLYGON; \quad r: RECTANGLE; \quad t: TRIANGLE; \]
\[ x: REAL \]

Permitted:

\[ x := p.\text{perimeter} \]
\[ x := r.\text{perimeter} \]
\[ x := r.\text{diagonal} \]
\[ p := r \]

NOT permitted:

\[ x := p.\text{diagonal} \] (even just after \( p := r \) !)
\[ r := p \]
Polymorphism and dynamic binding

- What is the effect of the following (assuming *some_test* is true)?

```java
if some_test then
    p := r
else
    p := t
end
x := p.perimeter
```

- **Redefinition**: A class may change an inherited feature, as with *RECTANGLE* redefining perimeter of *POLYGON*.

- **Polymorphism**: *p* may have different forms at run-time.

- **Dynamic binding**: Effect of *p.perimeter* depends on run-time form of *p*. 

---

*Chair of Software Engineering*  
Introduction to Programming – Lecture 18
Dynamic binding: Using non-O-O techniques

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

and similarly for all other routines!

Tedious; must be changed whenever there’s a new figure type.
Dynamic binding: In action

With:

\[\text{figure_list: LIST [FIGURE]}\]
\[c: \text{CIRCLE}\]
\[p: \text{POLYGON}\]
\[f: \text{FIGURE}\]

and:

\[\text{create c.make}\]
\[\text{create p.make}\]
\[\text{create figure_list.make}\]

Initialize:

\[\text{figure_list.extend (c)}\]
\[\text{figure_list.extend (p)}\]

Then just use:

\[f := \text{figure_list.i_th (i)}\]
\[f.move (...)\]
\[f.rotate (...)\]
\[f.display\]
\[-- \text{and so on for every}\]
\[-- \text{operation on } f!\]
Inheritance

Abstraction

SET_OF_BOOKS

LIST_OF_BOOKS

LIKED_LIST_OF_BOOKS

Specialization
Genericity: Type parametrization
Genericity vs. Inheritance

- List of books
- Set of books
- List of people
- List of journals
- Liked list of books

Abstraction

Type parameterization

Specialization
Genericity: \( \text{LIST} [G] \)

\textbf{class} \( \text{LIST} [G] \)

\textbf{feature}

\[
\begin{align*}
\text{... item}: \ G & \text{ is } \ldots \\
\text{extend} (x: \ G) & \text{ is } \ldots
\end{align*}
\]

\textbf{end}

\( \text{figure_list}: \ \text{LIST} [\text{FIGURE}] \)

\( r: \ \text{RECTANGLE} \)

\( s: \ \text{SQUARE} \)

\( t: \ \text{TRIANGLE} \)

\( p: \ \text{POLYGON} \)

\( \text{figure_list.extend} (p) \)

\( \text{figure_list.extend} (t) \)

\( \text{figure_list.extend} (s) \)

\( \text{figure_list.extend} (r) \)

\( \text{figure_list.i_th} (i).\text{display} \)
figure_list.store ("FILE_NAME")

...

-- Two years later:

\[
\begin{align*}
\text{figure_list} &:= \text{retrieved} ("FILE\_NAME") \\
x &:= \text{figure_list.i\_th} (i) \quad -- [1] \\
\text{print} (x.\text{diagonal}) &\quad -- [2]
\end{align*}
\]

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

\[ x \ ?= \ y \]

with

\[ x : A \]

- If \( y \) is attached to an object whose type conforms to \( A \), perform normal reference assignment.
- Otherwise, make \( x \) void.
Forcing a type: The Solution (using an assignment attempt)

\[ f: \text{FIGURE} \]
\[ r: \text{RECTANGLE} \]

... 

\[ \text{figure} \_\text{list} := \text{retrieved} \("\text{FILE\_NAME}\") \]
\[ f := \text{figure} \_\text{list}.i\_\text{th} (i) \]
\[ r ?= f \]

\[ \text{if } r /= \text{Void then} \]
\[ \quad \text{print} (r.\text{diagonal}) \]
\[ \text{else} \]
\[ \quad \text{print} ("\text{Too bad.}\") \]
\[ \text{end} \]
class VECTOR [G]

feature

infix "+" (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
Constrained genericity

- The body of **infix** ":+"::

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

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

- Declare class \textit{VECTOR} as

  
  \begin{verbatim}
  class VECTOR [G -> NUMERIC]
  feature
  \end{verbatim}

  ... The rest as before ...

  \end{verbatim}

- Class \textit{NUMERIC} (from the Kernel Library) provides features \texttt{infix} "+", \texttt{infix} "*" and so on.
Correct call:

```plaintext
if a1.α then
    a1.r (....)
else
    ...
end
```
Assertion redeclaration rule

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

\[
\begin{align*}
\text{require else } & \text{ new_pre} \\
\text{ensure then } & \text{ new_post}
\end{align*}
\]

- Resulting assertions are:
  - `original_precondition` or `new_pre`
  - `original_postcondition` and `new_post`
- Preconditions that are inherited can only be weakened
- Postconditions that are inherited can only be strengthened
Assertions: Invariant accumulation

- Every class inherits all the invariant clauses of its parents.
- These clauses are conceptually “and”-ed.