Introduction to Programming

Bertrand Meyer

Last revised 12 January 2004
Lecture 20:
More about inheritance
Topics

- Using deferred classes
- (Digression: stacks and abstract data types)
- Multiple inheritance
- The proper use of inheritance
A digression: stacks

Basic container structure:

Last-In, First-Out (LIFO)

“push” operation: feature *put*
Stacks

Basic container structure:

Last-In, First-Out (LIFO)

“pop” operation:
feature \textit{remove}
Stacks

Basic container structure:

Last-In, First-Out (LIFO)

“top” operation: feature item
Stack specified by mathematical functions

item (put (s, x)) = x

remove (put (s, x)) = s

\[
\text{put ( } \quad , \quad \text{ ) } = \quad \text{ } \quad \text{ } \quad \text{ } \quad \text{ } \\
\text{ } \quad \text{ } \quad \text{ } \quad \text{ } \\
S \quad X \quad S'
\]
Abstract Data Types (ADT)

Characterize data structures by a set of mathematical functions, and their abstract mathematical properties

The theoretical basis for object-oriented programming (and much of modern design and programming methodology)

\[ \text{item} \left( \text{put} \left( s, x \right) \right) = x \]

\[ \text{remove} \left( \text{put} \left( s, x \right) \right) = s \]
“Competitors” to stacks

- Queues: FIFO (first-in first-out)
- Priority queues
Use of stacks

- Compiling, especially parsing (syntax analysis)

\[ x \times (a + 2 \times b) \]

- Operating systems

- Implementing recursion (also part of compiling)
The recursive algorithm of Hanoi

move (n: INTEGER;
source, target, other: CHARACTER) is
  -- Move n disks from needle source to target,
  -- using other as storage.
  do
    if n > 0 then
      move (n−1, source, other, target)
      transfer (source, target)
      move (n−1, other, target, source)
    end
  end
Stack representation: array (up)

capacity  count  representation

“Push” operation:
Stack representation: array (down)

“Push” operation:

representation

capacity

free

1
Stack representation: Linked

“Push” operation:
The inheritance hierarchy

- Deferred class: STACK

- Effective classes: LINKED_STACK, ARRAYED_STACK etc.

*: deferred
The role of deferred classes

- Express abstract concepts independently of implementation
- Express common elements of various implementations
Deferred classes in EiffelBase

CONTAINER

BOX

* deferred

FINITE

INFINITE

* deferred

BOUNDED

UNBOUNDED

COUNTABLE

COLLECTION

BAG

SET

* deferred

HIERARCHICAL

LINEAR

TRAVERSABLE

CONTAINER

COLLECTION

* deferred

BOUNDED

UNBOUNDED

COUNTABLE

TABLE

ACTIVE

INFINITE

FINITE

* deferred

RESIZABLE

INDEXABLE

CURSOR_STRUCTURE

DISPENSER

SEQUENCE

STACK

QUEUE

* deferred

* : deferred
Multiple inheritance

- Allow a class to have two or more parents.
- Examples that come to mind: **ASSISTANT** inherits from **TEACHER** and **STUDENT**.

```
TEACHER

ASSISTANT

STUDENT
```
This is in fact a case of repeated inheritance:
Common examples of multiple inheritance

- Combining separate abstractions:
  - Restaurant, train car
  - Calculator, watch
  - Plane, asset
Multiple inheritance: Combining abstractions

**Infix operators:**
- `<`
- `>`
- `<=`
- `>=`

**Classes:**
- `COMPARABLE`
- `INTEGER`
- `REAL`
- `STRING`
- `DOUBLE`
- `COMPLEX`
- `NUMERIC`

**Operations:**
- `+`
- `-`
- `*`
- `/`

**Relationships:**
- `COMPARABLE` is related to `INTEGER`, `REAL`, `STRING`, and `DOUBLE`.
- `INTEGER` and `REAL` are related to `COMPARABLE`.
- `NUMERIC` is related to `COMPARABLE` and `DOUBLE`.
- `DOUBLE` and `COMPLEX` are related to `COMPARABLE`.

**Diagram:**
- A diagram illustrating the relationships and operations between the classes.

---

*Chair of Software Engineering*

*Introduction to Programming – Lecture 17*
Multiple inheritance: Nested windows

- "Graphical" features: height, width, change_height, change_width, xpos, ypos, move...
- "Hierarchical" features: superwindow, subwindows, change_subwindow, add_subwindow...

```plaintext
class WINDOW
  inherit RECTANGLE
    TREE [WINDOW]
  feature
    ...
end
```

```
Chair of Software Engineering
Introduction to Programming – Lecture 17
```
Multiple inheritance: Composite figures

Simple figures

A composite figure
Defining the notion of composite figure

* FIGURE

display

hide

rotate

move

...

LIST [FIGURE]
count

put

remove

...

COMPOSITE FIGURE
Defining the notion of composite figure through multiple inheritance

* 
FIGURE

OPEN FIGURE

SEGMENT

POLYLINE

POLYGON

TRIANGLE

SQUARE

CLOSED FIGURE

perimeter*

perimeter+

perimeter++

diagonal

POLYLINE

perimeter+

COMPOSITE FIGURE

LIST [FIGURE]

RECTANGLE

CIRCLE
A composite figure as a list

start

item

forth

after
Composite figures

class
   COMPOSITE FIGURE
inherit
   FIGURE
   redefine display, move, rotate, ...
end

LIST [FIGURE]

feature
   display is
   do
      -- Display each constituent figure in turn.
      from
      until
      loop
         item.display
         forth
      end
   end

   ... Similarly for move, rotate etc. ...
end
A simpler form of procedures *display*, *move* etc. can be obtained through the use of iterators.

We’ll learn to use *agents* for that purpose.
Name clashes under multiple inheritance

Diagram:

- Class A
- Class B
- Class C

foo

Relationships:
- A inherits from C
- B inherits from C
- foo is a method in both A and B
Resolving name clashes

rename `foo` as `fog`

rename `foo` as `zoo`
Resolving name clashes (cont’d)

class C inherit

A

rename foo as fog end

B

rename foo as zoo end

feature ...

Chair of Software Engineering
Results of renaming

\[
\begin{align*}
a1 &: A \\
b1 &: B \\
c1 &: C \\
... \\
c1.fog \\
c1.zoo \\
a1.foo \\
b1.foo \\
\end{align*}
\]

Invalid:
\[
\text{a1.fog, a1.zoo, b1.zoo, b1.fog, c1.foo}
\]
Another application of renaming

- Provide locally better adapted terminology.

- Example: *child (TREE)*; *subwindow (WINDOW)*.
Multiple inheritance: Nested windows

- "Graphical" features: height, width, change_height, change_width, xpos, ypos, move...
- "Hierarchical" features: superwindow, subwindows, change_subwindow, add_subwindow...

```plaintext
class WINDOW
  inherit
    RECTANGLE
    TREE [WINDOW]
  feature
    ...
  end
```
class ARRAYED_STACK [G] inherit
  STACK [G]
  ARRAY [G]
feature
  ...
end

class LINKED_STACK [G] inherit
  STACK [G]
  LINKED_LIST [G]
feature
  ...
end

This is controversial!
Others prefer to use the client relation
Using inheritance properly

Two relations: client, inheritance

- **Client** expresses that instances of $B$ must possess information about instances of $A$.

- **Inheritance** expresses that every instance of $D$ may be viewed as an instance of $C$. 

![Diagram showing client and inheritance relations between classes B, A, C, and D.]
From a widely used software engineering textbook:

Multiple inheritance allows several objects to act as base objects and is supported in object-oriented languages such as Eiffel (Meyer, 1988). The characteristics of several different object classes can be combined to make up a new object.

For example, say we have an object class CAR which encapsulates information about cars and an object class PERSON which encapsulates information about people.

We could use both of these to define a new object class CAR-OWNER which combines the attributes of CAR and PERSON. Adaptation through inheritance tends to lead to extra functionality being inherited, which can make components inefficient and bulky.
Inheritance?

PERSON

CAR

CAR_OWNER
Inheritance? ... or client?

Diagram:

- PERSON
- CAR
- CAR_OWNER
"He has a head like an Austin Mini with the doors open."

(From: *The Dictionary of Aussie Slang*, Five-Mile Press, Melbourne, Australia.)
Except for polymorphic uses, inheritance is never required:

- Rather than having \( B \) inherit from \( A \) you can always have \( B \) include an attribute of type \( A \) (or expanded \( A \)) – except if an entity of type \( A \) may have to represent values of type \( B \).
To be is also to have!

- (1) Every software engineer is an engineer.
- (2) Every software engineer has a part of himself which is an engineer.

But:

TO HAVE IS NOT ALWAYS TO BE!
Would you rather buy or inherit?

- A case in which having is not being (i.e. “client” is OK but not inheritance):
  - Every object of type $B$ has a component of type $A$, BUT that component may need to be replaced during the object’s lifetime.

- Use the client relation instead:

```plaintext
class WINDOW
inherit GENERAL_WINDOW
WINDOW_IMPLEMENTATION
feature ...
end
```
Handles (the bridge pattern)

```plaintext
class WINDOW inherit GENERAL_WINDOW

feature

handle: TOOLKIT

...

set_handle (t: TOOLKIT) is
  do
    handle := t
  end

...

end
```

```
display is
do
  handle.display (Current)
end
```

```
WINDOW
  TOOLTIP

GTK

MS WINDOWS

display*

display+

handle.display (Current)

...
More inheritance advice

- Avoid “taxomania”: don’t overclassify.

- Introduce a new class only if it corresponds to a meaningful abstraction, with its own features, usually queries and commands.

- When in doubt, wait until you are sure that a new class is needed. Adding a class is a significant design decision.
End of lecture 20