Object-Oriented Software Construction

Bertrand Meyer
Lecture 13:
Advanced inheritance mechanisms
Deferred classes

defered class STACK [G] feature

put (x: G) is
    -- Push x on top of stack.
    require
        not full
    deferred
    ensure
        item = x
        not empty
    end
The STACK class (cont’d)

remove is

-- Pop top element of stack.
require
  not empty
deferred
ensure
  not full
  count = old count – 1
end
The STACK class (cont’d)

**full**: BOOLEAN is

  -- Is there no room place for new items?
  deferred
  end

**empty**: BOOLEAN is

  -- Is there no item?
  deferred
  end

**count**: BOOLEAN is

  -- Number of items on stack
  deferred
  ensure
      Result >= 0
  end
The STACK class (cont’d)

-- Not all features need be deferred!

\textit{replace} \ (x: \ G) \ is \\
\hspace{1em} -- Replace top element of stack by $x$.

\textbf{require} \\
\hspace{1em} \textbf{not} \ \textbf{empty}

\textbf{do} \\
\hspace{1em} \textit{remove} \\
\hspace{2em} \textit{put} \ (x)

\textbf{ensure} \\
\hspace{1em} \textbf{not} \ \textbf{empty} \\
\hspace{2em} \textit{item} = x \\
\hspace{2em} \textit{count} = \textbf{old} \ \textit{count}

\textbf{end}

\textbf{invariant}

\hspace{1em} \textbf{not} \ (\textbf{full} \ \textbf{and} \ \textbf{empty})

\textbf{end}
Applications of deferred classes

- Taxonomy
- Library organization
- Capturing common abstractions
- Capturing common behaviors
- “Don’t call us, we’ll call you”
- Analysis
- High-level architecture and design
EiffelBase: Container structures

CONTAINER
  * BOX
  * CONTAINER
  * COLLECTION
  * TRAVERSABLE
  * BAG
  * SET
  * HIERARCHICAL
  * LINEAR
  * BAG
  * SET
  * HIERARCHICAL
  * LINEAR
  * TABLE
  * ACTIVE
  * INTEGRAL INTERVAL
  * BILINEAR
  * CURSOR STRUCTURE
  * DISPENSER
  * SEQUENCE
  * ARRAY
  * STRING
  * HASH_TABLE
  * STACK
  * QUEUE
  * FINITE
  * INFINITE
  * BOUNDED
  * UNBOUNDED
  * COUNTABLE
  * INDEXABLE
  * RESIZABLE
  * INDEXABLE
  * CURSOR STRUCTURE
  * DISPENSER
  * SEQUENCE
  * ARRAY
  * STRING
  * HASH_TABLE
  * STACK
  * QUEUE
  * COUNTABLE
  * INFINITE
  * BOUNDED
  * UNBOUNDED
  * COUNTABLE
  * INDEXABLE
  * RESIZABLE
  * INDEXABLE
  * CURSOR STRUCTURE
  * DISPENSER
  * SEQUENCE
deferred class VAT inherit TANK

feature

in_valve, out_valve: VALVE

fill is

    -- Fill the vat.
    require
    in_valve.open
    out_valve.closed

    deferred
    ensure
    in_valve.closed
    out_valve.closed
    is_full

end

empty, is_full, is_empty, gauge, maximum, ... [Other features] ...

invariant

    is_full = (gauge >= 0.97 * maximum) and (gauge <= 1.03 * maximum)

end
Indirect and direct repeated inheritance

A

B

C

D

A

D
Repeated inheritance

- Assume class `TAXPAYER` with attributes
  
  ```
  age: INTEGER  
  address: STRING  
  bank_account: ACCOUNT  
  tax_id: INTEGER  
  ```

- and routines such as
  
  ```
  pass_birthday is  
  do  
    age := age + 1  
  end  
  pay_taxes is ...  
  deposit_to_account (sum: INTEGER) is ...
  ```
Repeated inheritance (cont’d)

- Heirs may include \textit{SWISS\_TAXPAYER} and \textit{US\_TAXPAYER}.

\begin{itemize}
\item \textit{TAXPAYER} \texttt{age} \texttt{address} \texttt{tax\_id} \texttt{pass\_birthday} \texttt{pay\_taxes}
\item \textit{US\_TAXPAYER}
\item \textit{SWISS\_TAXPAYER}
\end{itemize}
Repeated inheritance (cont’d)

- The two above classes may in turn have a common heir: **SWISS_US_TAXPAYER**.
Repeated inheritance issues

- What happens with features inherited twice from the common ancestor `TAXPAYER`, such as `address`, `age`, `tax_id`, `pass_birthday`?
The inheritance clause

```
inherit

SWISS_TAXPAYER
  rename
    address as swiss_address,
    tax_id as swiss_tax_id,
    pay_taxes as pay_swiss_taxes,
    bank_account as swiss_bank_account,
    deposit_to_account as deposit_to_swiss_account,
  ...
end

US_TAXPAYER
  rename
    address as us_address,
    tax_id as us_tax_id,
    pay_taxes as pay_us_taxes,
    bank_account as us_bank_account,
    deposit_to_account as deposit_to_us_account,
  ...
end
```
Sharing and replication

- Features such as `age` and `birthday`, not renamed along any of the inheritance paths, will be shared.
- Features such as `tax_id`, inherited under different names, will be replicated.
The need for select

- Assume there is a redefinition somewhere along the way:
A potential ambiguity arises because of polymorphism and dynamic binding:

\[
\begin{align*}
t &: \text{TAXPAYER} \\
s &: \text{SWISS\_TAXPAYER} \\
u &: \text{US\_TAXPAYER} \\
su &: \text{SWISS\_US\_TAXPAYER}
\end{align*}
\]

\[
\begin{align*}
\text{if } \ldots \text{ then} \\
\quad t &= s \\
\text{else} \\
\quad t &= su \\
\text{end} \\
\ldots \\
\text{print} \ (t.\text{address})
\end{align*}
\]
Removing the ambiguity

```java
class SWISS_US_TAXPAYER inherit SWISS_TAXPAYER
  rename
    address as swiss_address,
    tax_id as swiss_tax_id,
    pay_taxes as pay_swiss_taxes,
    bank_account as swiss_bank_account,
    deposit_to_account as deposit_to_swiss_account,
  ...
  select
    swiss_address,
    swiss_tax_id,
    pay_swiss_taxes,
    swiss_bank_account,
    deposit_to_swiss_account
  end

US_TAXPAYER
  rename
    address as us_address,
    tax_id as us_tax_id,
  ...
  end
```
Creating with a specified type

- To avoid this:

  \[
  \begin{align*}
  a1 &: A \\
  b1 &: B \\
  \ldots \\
  \text{create } & b1.\text{make } (...) \\
  a1 &:= b1
  \end{align*}
  \]

- Simply use

  \[
  \begin{align*}
  a1 &: A \\
  \ldots \\
  \text{create } & \{B\} \ a1.\text{make } (...) 
  \end{align*}
  \]
Once routines

- If instead of
  
  \[
  r \text{ is} \\
  \text{do} \\
  \text{end}
  \]
  
  ... Instructions ...

- you write

  \[
  r \text{ is} \\
  \text{once} \\
  \text{end}
  \]
  
  ... Instructions ...

- then *Instructions* will be executed only for the first call by any client during execution. Subsequent calls return immediately.

- In the case of a function, subsequent calls return the result computed by the first call.
Scheme for shared objects

class \textit{SHARED\_OBJECTS} feature

\hspace{1em} error\_window: \textit{WINDOW} is
\hspace{2em} once
\hspace{3em} create Result.make (...) 
\hspace{2em} end

\hspace{1em} exit\_button: \textit{BUTTON} is
\hspace{2em} once
\hspace{3em} create Result.make (...) 
\hspace{2em} end

\hspace{1em} ...
\hspace{1em} end

end

\textbf{class} \textit{MY\_APPLICATION\_CLASS} \textbf{inherit} \textit{SHARED\_OBJECTS}

feature

\hspace{2em} r is
\hspace{3em} do
\hspace{4em} error\_window.put (my\_error\_message)
\hspace{3em} end

\hspace{2em} ...

end
Undefining a feature

```plaintext
defferred class B inherit
  A
  undefine f
end

feature
  ...
end
```
Feature merging

\[ f^* \quad A \quad f^* \quad B \quad f^+ \quad C \]

\[ D \]
Feature merging (cont’d)

class D inherit

  A
  B
  C

feature

  ...

end
Feature merging: with different names

A

B

C

D

g*  f*
h+
g f

h f
Feature merging: with different names

```plaintext
class D inherit
    A
        rename
            g as f
        end
    end
B
C
    rename
        h as f
    end
feature
    ...
end
```

Chair of Software Engineering
Feature merging: effective features

\[ g^* \rightarrow A \]
\[ f^* \rightarrow B \]
\[ h^+ \rightarrow C \]

\[ g \mathbin{\uparrow} f \quad f^- \]
\[ h \mathbin{\uparrow} f \quad f^- \]

\[ a1: A \quad b1: B \quad c1: C \quad d1: D \]
\[ a1.g \quad b1.f \quad c1.h \quad d1.f \]
class $D$ inherit $A$
    rename $g$ as $f$
    undefine $f$
end

$B$

$C$
    rename $h$ as $f$
    undefine $f$
end

feature ...
end
When is a name clash acceptable?

- (Between n features of a class, all with the same name, immediate or inherited.)
  - They must all have compatible signatures.
  - If more than one is effective, they must all come from a common ancestor feature under repeated inheritance.
Feature adaptation clauses

- rename
- export
- undefine
- redefine
- select
Export adaptation

class $B$ inherit $A$

    export

    {ANY} all
    {NONE} $h$
    {$A, B, C, D$} $i, j, k$

end

feature

...
End of lecture 13