Programming in the large

Bertrand Meyer
Lecture 11: Multiple inheritance
Agenda for today

- Using inheritance properly
- Contracts and inheritance
- Feature merging and adaptation clauses
Agenda for today

- Using inheritance properly
  - Contracts and inheritance
  - Feature merging and adaptation clauses
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 $C$. 
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?

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.)
Because polymorphic uses:

- 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.

### Diagram:

- B
- (B)
- (A)
(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
```
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.
Agenda for today

- Using inheritance properly
- Contracts and inheritance
- Feature merging and adaptation clauses
Contracts and inheritance

- **Issues:** what happens, under inheritance, to
  - Class invariants?
  - Routine preconditions and postconditions?
Invariants

- Invariant Inheritance rule:
  - The invariant of a class automatically includes the invariant clauses from all its parents, "and"-ed.
  - Accumulated result visible in flat and interface forms.
Contracts and inheritance

Correct call:

\[
\text{if } a1.\alpha \text{ then} \\
\text{a1.r (\ldots)} \\
\text{-- Here } a1.\beta \text{ holds.} \\
\text{end}
\]
Assertion redeclaration rule

- When redeclaring a routine:
  - Precondition may only be kept or weakened.
  - Postcondition may only be kept or strengthened.

- Redeclaration covers both redefinition and effecting.

- Should this remain a purely methodological rule? A compiler can hardly infer e.g. that:

  \[ n > 1 \]

  implies (is stronger) than

  \[ n^{26} + 3 \times n^{25} > 3 \]
A simple language rule does the trick!

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:

- **new_pre** or else **original_precondition**
- **original_postcondition** and then **new_post**
Agenda for today

- Using inheritance properly
- Contracts and inheritance
- Feature merging and adaptation clauses
Feature merging
Undefining a feature

defered class  

B

inherit A

undefine f

class A

feature

... 

class B extends A

define f 

end
Feature merging

class $D$
inherit $A$
inherit $B$
inherit $C$

feature

...
Feature merging: with different names

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

\[ h \mapsto f \]
Feature merging: with different names

class D
inherit A
  rename g as f
end

C
  rename h as f
end

feature ...
end
Feature merging: effective features

A

B

C

D

\[ g^* \]
\[ f^* \]
\[ h^+ \]

\[ g \leftarrow f \]
\[ f \rightarrow g \]
\[ h \leftarrow f \]
\[ f \rightarrow h \]

\[ a1: A \]
\[ b1: B \]
\[ c1: C \]
\[ d1: D \]
\[ a1.g \]
\[ b1.f \]
\[ c1.h \]
\[ d1.f \]
Feature merging: effective features

class $D$
  inherit $A$
    rename $g$ as $f$
    undefine $f$
  end

$B$

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

feature ...
end
Indirect and direct repeated inheritance

A

B

C

D

A

D
Multiple is also repeated inheritance

A typical case:

A typical case: A, B, and C have a common ancestor ANY. The inheritance relationships are:

- **A** inherits from **ANY** and other classes, and has methods `copy++` and `is_equal++`.
- **B** inherits from **ANY** and other classes, and has methods `copy` and `is_equal`.
- **C** has no direct inheritance from **ANY** but is related to **A** and **B** through unspecified relationships.

The diagram illustrates the inheritance structure, with arrows indicating inheritance relationships.
Repeated inheritance

- Assume class **TAXPAYER** with attributes
  
  \[
  \begin{align*}
  \text{age: } & \text{INTEGER} \\
  \text{address: } & \text{STRING} \\
  \text{bank_account: } & \text{ACCOUNT} \\
  \text{tax_id: } & \text{INTEGER}
  \end{align*}
  \]

- and routines such as
  
  \[
  \begin{align*}
  \text{pass_birthday is} & \\
  \text{do} & \\
  \text{age := age + 1} & \\
  \text{end} & \\
  \text{pay_taxes is} & \ldots \\
  \text{deposit_to_account (sum: INTEGER) is} & \ldots
  \end{align*}
  \]
Repeated inheritance

- Heirs may include `SWISS_TAXPAYER` and `US_TAXPAYER`.

```
TAXPAYER
  age
  address
  tax_id
  pass_birthday
  pay_taxes

US_TAXPAYER

SWISS_TAXPAYER
```
Repeated inheritance

- 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:

```
TAXPAYER
  address

US_TAXPAYER
  address++

SWISS_TAXPAYER
  address++

SWISS_US_TAXPAYER
  address  us_address
  address  swiss_address
```
The need for select

- A potential ambiguity arises because of polymorphism and dynamic binding:

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

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

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
Feature adaptation clauses

- rename
- export
- undefine
- redefine
- select
class B
inherit A
export
  {ANY} all
  {NONE} h
  {A, B, C, D} i, j, k
end

feature...
...
End lecture 11