Programming in the large

Bertrand Meyer

Lecture 11: Multiple inheritance

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 \( C \).

Not the best use in the world...

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?

PERSON  CAR  CAR_OWNER

The 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.)

Client or inheritance?

- 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:
    ```
    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

**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
  
  \[ n^{26} + 3 * n^{25} > 3 \]
Assertion redeclaration rule in Eiffel

- A simple language rule does the trick!
- Redefined version may **not** have `require` or `ensure`.
- May have nothing (assertions kept by default), or
  
  ```eiffel
to require else new_pre
  to 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

```
class D
  inherit A
  B
  C
  feature ...
end
```

---

Undefining a feature

```eiffel
defined class B
inherit A
  define undefine f
end
feature ...
end
```

---

Feature merging: with different names

```
class D
  inherit A
  B
  C
  feature ...
end
```
### Feature merging: with different names

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

### Feature merging: effective features

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

```
\[
g' A r B c C h\       f h\      f\       f\       h
\]
```

### Feature merging: effective features

```plaintext
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

```plaintext
A B C D
A D
```

### Multiple is also repeated inheritance

- A typical case:

```
ANY copy is_equal
A B
  copy++ is_equal++
C ??
```

### Repeated inheritance

- Assume class `TAXPAYER` with attributes
  - `age: INTEGER`
  - `address: STRING`
  - `bank_account: ACCOUNT`
  - `tax_id: INTEGER`
- and routines such as
  - `pass_birthday` is
    ```plaintext
do age := age + 1
  end
```

  - `pay_taxes` is ...
  - `deposit_to_account (sum: INTEGER)` is ...

```
TAXPAYER address tax_id
pay_taxes
```
Repeated inheritance

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

Repeated inheritance issues

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

The inheritance clause

```plaintext
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:
The need for select

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

```plaintext
if ... then
  t := s
else
  t := su
end
```

print(t.address)

Removing the ambiguity

```plaintext
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
```

Export adaptation

```plaintext
class
  B
  inherit A
  export
    {ANY} all
    {NONE} h
    (A, B, C, D) i, j, k
end
```

Feature adaptation clauses

- rename
- export
- undefine
- redefine
- select

End lecture 11