Combining abstractions

Given the classes

- \texttt{TRAIN\_CAR}, \texttt{RESTAURANT}

how would you implement a \texttt{DINER}?

Examples of multiple inheritance

Combining separate abstractions:

- Restaurant, train car
- Calculator, watch
- Plane, asset
- Home, vehicle
- Tram, bus
Composite figures

A composite figure

Multiple inheritance: Composite figures

Simple figures

A composite figure

Defining the notion of composite figure

center display hide rotate move...

FIGURE

LIST [FIGURE]
count put remove...

COMPOSITE FIGURE
In the overall structure

A composite figure as a list

Composite figures

```
class COMPOSITE_FIGURE inherit FIGURE
   LIST[FIGURE]
   feature
display
   do -- Display each constituent figure in turn.
      from start until after loop
         item.display
         forth
   end
end
... Similarly for move, rotate etc. ...
```
Going one level of abstraction higher

A simpler form of procedures display, move etc. can be obtained through the use of iterators

Use agents for that purpose

Multiple inheritance: Combining abstractions

The Java-C# solution

No multiple inheritance for classes

"Interfaces": specification only (but no contracts)
  > Similar to completely deferred classes (with no effective feature)

A class may inherit from:
  > At most one class
  > Any number of interfaces
Multiple inheritance: Combining abstractions

\[ \leq, \leq_c, \geq, \geq_c, \ldots \]
(total order relation)

\[ \text{(commutative ring)} \]

COMPARABLE

NUMERIC

STRING

COMPLEX

INTEGER

REAL

\(<, \leq, >, \geq, \ldots \)

\(+, -, \times, /, \ldots \)

(total order relation)

COMMUTATIVE RING

How do we write COMPARABLE?

defered class COMPARABLE[G] feature

less alias "<" (x: COMPARABLE[G]): BOOLEAN deferred end

less_equal alias "\leq" (x: COMPARABLE[G]): BOOLEAN do

Result := (Current < x or (Current = x))
end

greater alias ">" (x: COMPARABLE[G]): BOOLEAN do

Result := (x < Current)
end

greater_equal alias "\geq" (x: COMPARABLE[G]): BOOLEAN do

Result := (x <= Current)
end

Lessons from this example

Typical example of program with holes

We need the full spectrum from fully abstract (fully deferred) to fully implemented classes

Multiple inheritance is there to help us combine abstractions
Multiple inheritance: Name clashes

Resolving name clashes

Consequences of renaming

\[
\begin{align*}
\text{Consequences of renaming:} & \\
a1: & A \\
b1: & B \\
c1: & C \\
& \ldots \\
c1.f & \\
c1.A_f & \\
a1.f & \\
b1.f & 
\end{align*}
\]
Are all name clashes bad?

A name clash must be removed unless it is:
- Under repeated inheritance (i.e. not a real clash)
- Between features of which at most one is effective (i.e. others are deferred)

Feature merging

Feature merging: with different names
Feature merging: effective features

Merging through undefined
Merging effective features with different names

Acceptable name clashes

If inherited features have all the same names, there is no harmful name clash if:
- They all have compatible signatures
- At most one of them is effective

Semantics of such a case:
- Merge all features into one
- If there is an effective feature, it imposes its implementation

Feature merging: effective features
A special case of multiple inheritance

Allow a class to have two or more parents.

Examples that come to mind: ASSISTANT inherits from TEACHER and STUDENT.

This is in fact a case of repeated inheritance

Indirect and direct repeated inheritance

Multiple is also repeated inheritance

A typical case:
Acceptable name clashes

If inherited features have all the same names, there is no harmful name clash if:
- They all have compatible signatures
- At most one of them is effective

Semantics of such a case:
- Merge all features into one
- If there is an effective feature, it imposes its implementation

Sharing and replication

Features such as \( f \), not renamed along any of the inheritance paths, will be shared.
Features such as \( g \), inherited under different names, will be replicated.

The need for select

A potential ambiguity arises because of polymorphism and dynamic binding:

\[
a : \text{ANY} \\
d1 : D \\
\ldots \\
\]

\[
a := d1 \\
a\text{.copy}(\ldots) \\
\]

\[
a : \text{ANY} \\
copy \\
is\_equal \\
\]

\[
\text{LIST} \\
copy++ \\
is\_equal++ \\
\]

\[
\text{C} \\
copy \Rightarrow \text{C}\_copy \\
is\_equal \Rightarrow \text{C}\_is\_equal \\
\]

\[
\text{D} \\
copy \Rightarrow \text{C}\_copy \\
is\_equal \Rightarrow \text{C}\_is\_equal \\
\]
Removing the ambiguity

```ruby
class D
  inherit LIST [T]
  select copy, is_equal
end

C
  rename copy as C_copy, is_equal as C_is_equal,
  ... 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.

Another application of renaming

Provide locally better adapted terminology.

Example: `child(TREE); subwindow(WINDOW)`.
What we have seen

A number of games one can play with inheritance:

- Multiple inheritance
- Feature merging
- Repeated inheritance

End of lecture 21