Lecture 21
Multiple inheritance
Combining abstractions

Given the classes

- TRAIN_CAR, RESTAURANT

how would you implement a DINER?
Examples of multiple inheritance

Combining separate abstractions:

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

Simple figures

A composite figure
Defining the notion of composite figure

FIGURE

LIST [FIGURE]

COMPOSITE FIGURE

center
display
hide
rotate
move...

count
put
remove...

Introduction to Programming, lecture 21: Multiple inheritance
In the overall structure

- FIGURE
- LIST
- COMPOSITE FIGURE

- OPEN FIGURE
- CLOSED FIGURE

- POLYGON
  - perimeter+
  - diagonal

- ELLIPSE
  - perimeter+

- RECTANGLE
  - perimeter++

- SQUARE
  - perimeter++

- POLYLINE

- SEGMENT

- TRIANGLE
  - perimeter++
A composite figure as a list

item

forth

Cursor

after
Composite figures

```plaintext
class COMPOSITE_FIGURE inherit FIGURE
    LIST[FIGURE]

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

... Similarly for move, rotate etc. ...
```

Requires dynamic binding
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

- **COMPARABLE**
  - `<`, `<=`, `>`, `>=`, ...
  - (total order relation)

- **NUMERIC**
  - `<`, `<=`, `>`, `>=`, `+`, `-`, `*`, `/`
  - (commutative ring)

- **INTEGER**

- **REAL**

- **STRING**

- **COMPLEX**

Introduction to Programming, lecture 21: Multiple inheritance
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

- COMPARABLE
- NUMERIC
- INTEGER
- REAL
- STRING
- COMPLEX

(<, <=, >, >=, ...
(total order relation)

+(+, –, *, /
(commutative ring)
How do we write COMPARABLE?

defferred class COMPARABLE[\text{G}] feature

\text{less} alias "<" (x: COMPARABLE[\text{G}]): BOOLEAN

defferred
end

\text{less_equal} alias "\leq" (x: COMPARABLE[\text{G}]): BOOLEAN

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

greater alias ">
  (x: COMPARABLE[\text{G}]): BOOLEAN

do Result := (x < Current) end

greater_equal alias ">=\) (x: COMPARABLE[\text{G}]): BOOLEAN

do Result := (x \leq 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

Diagram: A and B both inherit from C, but the names are the same.

Introduction to Programming, lecture 21: Multiple inheritance
Resolving name clashes

rename f as A_f
Consequences of renaming

\[ a1: A \]
\[ b1: B \]
\[ c1: C \]
...
\[ c1.f \]
\[ c1.A_f \]
\[ a1.f \]
\[ b1.f \]

**Invalid:**
- \[ a1.first_f \]
- \[ b1.first_f \]
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

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

D

Deferred
Effective
Feature merging: with different names

```
class D inherit A 
    rename g as f 
end 
B 
C 
    rename h as f 
end 
feature ...
end
```
Feature merging: effective features

A

B

C

D

* Deferred
+ Effective
-- Undefine
deferred class $T$
  inherit $S$
  undefine $v$
end

feature
  ...
end
Merging through undefinedation

class D
inherit A
  undefine f end
B
C
  undefine f end
feature ...
end
Merging effective features with different names

\[
\begin{align*}
A & \quad f^+ \\
B & \quad g^+ \\
C & \quad h^+ \\
D & \quad f^{-}
\end{align*}
\]

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

Introduction to Programming, lecture 21: Multiple inheritance
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

\[ g^+ \quad A \quad f^+ \quad B \quad h^+ \quad C \]

\[ g \rightarrow f \quad f \quad h \rightarrow f \]

a1: A  b1: B  c1: C  d1: D
a1.g  b1.f  c1.h  d1.f
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

A
B
C
D

A
D

Introduction to Programming, lecture 21: Multiple inheritance
Multiple is also repeated inheritance

A typical case:

```
copy++
is_equal++
```

```
ANY
  copy
  is_equal
```

```
LIST
```

```
C
```

```
D
  copy
  C_copy
  is_equal
  C_is_equal
```

```
??
```
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:

\[ a \colon \text{ANY} \]
\[ d1 \colon D \]

...  

\[ a := d1 \]

\[ a.copy (...) \]
Removing the ambiguity

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: \textit{child (TREE)}; \textit{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