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

Composite figures

Multiple inheritance: Composite figures
Defining the notion of composite figure

Composite figures through multiple inheritance

A composite figure as a list
Composite figures

```java
class COMPOSITE_FIGURE inherit FIGURE
    redefine display, move, rotate, ...
end

LIST[FIGURE]
feature
display is
    do
        -- Display each constituent figure in turn.
        from start until after loop
            item.display
        end
    end
    forth
end

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

Complex figures

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

We'll learn to use `agents` for that purpose.

Multiple inheritance: Combining abstractions

```
COMPARABLE

NUMERIC

INTEGER

REAL

STRING

DOUBLE

COMPLEX
```

A graph showing the inheritance hierarchy of various numeric types such as `COMPARABLE`, `NUMERIC`, `INTEGER`, `REAL`, `STRING`, `DOUBLE`, and `COMPLEX`.
deferred class COMPARABLE [G] feature
  infix "<" (other: COMPARABLE [G]): BOOLEAN is
defered
end

infix "<=" (other: COMPARABLE [G]): BOOLEAN is
do
  Result := Current < other or equal (Current, other)
end

infix ">=" (other: COMPARABLE [G]) is ...
infix ">" (other: COMPARABLE [G]) is ...
...
end

Lessons from this example

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 of name clashes]
Resolving name clashes

A
rename foo as fog
B
rename foo as zoo
C

Results of renaming

a1: A
b1: B
c1: C
...
c1.fog
c1.zoo
al.foo
bl.foo

Invalid:
al.fog, al.zoo, bl.zoo, bl.fog, cl.foo

Feature merging
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: with different names

```ruby
class D
  inherit A
  rename g as f end

  B

  C

  rename h as f end

feature ... end
```
Feature merging: effective features

Undefining a feature

defered class
    T
    inherit
        S
        undefine
            v
            end
        feature
            ...
        end
    end

Feature merging: effective features

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

A special case of multiple inheritance

Indirect and direct repeated inheritance
Multiple is also repeated inheritance

A typical case:

Repeated inheritance

Assume class TAXPAYER with attributes

```plaintext
age: INTEGER
address: STRING
bank_account: ACCOUNT
tax_id: INTEGER
```

and routines such as

```plaintext
pass_birthday is
do age := age + 1
end
pay_taxes is ...
deposit_to_account (sum: INTEGER) is ...
```

Repeated inheritance

Heirs may include SWISS_TAXPAYER and US_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**?

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 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
```

The need for select

Assume there is a redefinition somewhere along the way:

```
TAXPAYER address

address++
TAXPAYER address++

US_TAXPAYER address

address ~ us_address

SWISS_US_TAXPAYER address

address ~ swiss_address
```

The need for select

A potential ambiguity arises because of polymorphism and dynamic binding:

```
t:: TAXPAYER
su:: SWISS_US_TAXPAYER
...

t := su
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
end
```

US_TAXPAYER

```plaintext
rename
  address as uk_address,
  tax_id as uk_tax_id
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).
Project Presentation

Organization

You can either work alone or team up with another student
Team of 2 people from the same exercise group
Team members should have similar programming experience

Content

Extension to Traffic
You choose what this extension is
Examples:
- Route planner
- Location information
- Construction site marking
- Avoiding tram collisions
- Traffic map editor
Tasks

1. Initial description of the idea
2. Project description, analysis, and design
3. Implementation and documentation

Initial description of the idea

A few sentences about what you want to do in the project
Email to your assistant

Deadline: 21 December 2006

Project description, analysis and design

Submit:
- Project description
  - A detailed description of the requirements
  - A clear statement of how the work will be divided between the 2 group members (if applicable)
- Analysis and design document
  - A short report describing the overall architecture of your system (including a BON class diagram)

Deadline: 8 January 2007
Implementation and documentation

Implement and test your application

Submit:
  ➢ Your code
    • Only .e files and project ecf file
    • Only relative paths in the ecf file
  ➢ Documentation: developer guide
    • Can reuse parts of the design document

Deadline: 22 January 2007

Presentations of the project

In the exercise sessions:
  23 January or (for Knuth group) 25 January

In the lecture:
  In the last lecture of the semester (30 January 2007)

Submission procedure

Everything goes on the wiki:
  ➢ Initial idea - as inline text on the page of your project
  ➢ Project description, design document, and developer guide - as uploaded PDF files
  ➢ Source code - as uploaded .zip file
Assessment criteria

Design
- Extendibility
- Ease of use

Functionality
- Does the implementation satisfy the specification

Quality of contracts
- Preconditions
- Postconditions
- Class invariants
- Loop invariants and variants

Assessment criteria (continued)

Documentation
- Project description
- Design document
- Developer guide

Quality of code
- Style guidelines
- Quality of code

Effort devoted to the project

End of lecture 16