Lecture 12: Introduction to inheritance and genericity
On the menu for today (& next time)

Two fundamental mechanisms for expressiveness and reliability:

- Genericity
- Inheritance

with associated (just as important!) notions:

- Static typing
- Polymorphism
- Dynamic binding
Extending the basic notion of class

- Abstraction
- Inheritance
- Genericity
- Type parameterization
- Specialization
Extending the basic notion of class

LIST_OF_CARS

SET_OF_CARS

SET_OF_PERSONS

LIST_OF_CITIES

LIST_OF_CARS

LIST_OF_PERSONS

LINKED_LIST_OF_CITIES

LINKED_LIST_OF_CARS

Inheritance

Genericity
Genericity

Unconstrained

\[ \text{LIST}[G] \]

- e.g. \text{LIST}[\text{INTEGER}], \text{LIST}[\text{PERSON}]

Constrained

\[ \text{HASH\_TABLE}[G \rightarrow \text{HASHABLE}] \]

\[ \text{VECTOR}[G \rightarrow \text{NUMERIC}] \]
Genericity: ensuring type safety

How can we define consistent “container” data structures, e.g. list of accounts, list of points?

Without genericity, something like this:

```plaintext
c : CITY ; p : PERSON
cities : LIST ...
people : LIST ...
```

What if wrong?

```plaintext
people.extend ( p )
cities.extend ( c )
c := cities.last
c.some_city_operation
```
Possible approaches

1. Duplicate code, manually or with help of macro processor

2. Wait until run time; if types don’t match, trigger a run-time failure (Smalltalk)

3. Convert (“cast”) all values to a universal type, such as “pointer to void” in C

4. Parameterize the class, giving an explicit name $G$ to the type of container elements. This is the Eiffel approach, also found in recent versions of Java, .NET and others.
A generic class

class LIST[\textbf{G}] feature

\hspace{1em} extend (x: G) ... \\
\hspace{1em} last: G ... \\
end

To use the class: obtain a \textit{generic derivation}, e.g.

cities: LIST[\textbf{CITY}]
Using generic derivations

\[
\begin{align*}
\text{cities} & : \text{LIST}[\text{CITY}] \\
\text{people} & : \text{LIST}[\text{PERSON}] \\
\text{c} & : \text{CITY} \\
\text{p} & : \text{PERSON} \\
\end{align*}
\]

... 

\[
\begin{align*}
\text{cities}.\text{extend} \ (\text{c}) \\
\text{people}.\text{extend} \ (\text{p}) \\
\end{align*}
\]

\[
\begin{align*}
\text{c} & := \text{cities}.\text{last} \\
\text{c}.\text{some}_\text{city}_\text{operation} \\
\end{align*}
\]

**STATIC TYPING**

The compiler will reject:

- \text{people}.\text{extend} \ (\text{c})
- \text{cities}.\text{extend} \ (\text{p})
Static typing

**Type-safe call** (during execution):

A feature call $x.f$ such that the object attached to $x$ has a feature corresponding to $f$

[Generalizes to calls with arguments, $x.f(a, b)$]

**Static type checker:**

A program-processing tool (such as a compiler) that guarantees, for any program it accepts, that any call in any execution will be **type-safe**

**Statically typed language:**

A programming language for which it is possible to write a **static type checker**
Using genericity

\[\text{LIST \ [CITY]}\]
\[\text{LIST \ [LIST \ [CITY]]}\]

A type is no longer exactly the same thing as a class!

(But every type remains \textbf{based} on a class.)
What is a type?

(To keep things simple let’s assume that a class has zero or one generic parameter)

A **type** is of one of the following two forms:

- **C**, where **C** is the name of a **non-generic class**
- **D [T]**, where **D** is the name of a **generic class** and **T** is a **type**
A generic class

class \texttt{LIST}\left[ G \right] \text{ feature}
\begin{align*}
\text{extend} (x : G) & \quad \ldots \\
\text{last} : G & \quad \ldots
\end{align*}
end

To use the class: obtain a \textcolor{red}{\textit{generic derivation}}, e.g.

\texttt{cities} : \texttt{LIST}\left[ \text{\textsc{city}} \right]
Reminder: the dual nature of classes

A class is a module
A class is a type*

As a module, a class:
- Groups a set of related services
- Enforces information hiding (not all services are visible from the outside)
- Has clients (the modules that use it) and suppliers (the modules it uses)

As a type, a class:
- Denotes possible run-time values (objects & references), the instances of the type
- Can be used for declarations of entities (representing such values)

*Or a type template (see genericity)
Reminder: how the two views match

The class, viewed as a *module*, groups a set of services (the *features* of the class) which are precisely the operations applicable to instances of the class, viewed as a *type*.

(Example: class *BUS*, features *stop, move, speed, passenger_count*)
Extending the basic notion of class

- **Abstraction**
  - **LIST_OF_CARS**
    - **SET_OF_CARS**
    - **LINKED_LIST_OF_CARS**

- **Inheritance**
  - **LIST_OF_CITIES**
  - **LIST_OF_PERSONS**

- **Genericity**
  - **Type parameterization**

- **Specialization**
Inheritance basics

Principle:
Describe a new class as extension or specialization of an existing class
(or several with *multiple* inheritance)

If $B$ inherits from $A$:

- **As modules**: all the services of $A$ are available in $B$
  (possibly with a different implementation)

- **As types**: whenever an instance of $A$ is required, an instance of $B$ will be acceptable
  ("is-a" relationship)
Terminology

If $B$ inherits from $A$ (by listing $A$ in its \textit{inherit} clause):

- $B$ is an \textit{heir} of $A$
- $A$ is a \textit{parent} of $B$

For a class $A$:

- The \textit{descendants} of $A$ are $A$ itself and (recursively) the descendants of $A$’s heirs
- \textit{Proper descendants} exclude $A$ itself

Reverse notions:

- Ancestor
- \textit{Proper ancestor}

More precise notion of instance:

- \textit{Direct instances} of $A$
- \textit{Instances} of $A$: the direct instances of $A$ and its descendants

(Other terminology: \textit{subclass}, \textit{superclass}, \textit{base class})
Example hierarchy (from Traffic)

* Deferred
+ Effective
++ Redefined

**MOVING**
- position
- update_coordinates
  - move

**VEHICLE**
- load

**TAXI**
- busy
- take*
- take+

**LINE_VEHICLE**
- update_coordinates**
  - move**

**EVENT_TAXI**

**TRAM**

**BUS**
Features in the example

Feature

take (from_location, to_location: COORDINATE)
  -- Bring passengers
  -- from from_location
  -- to to_location.

busy: BOOLEAN
  -- Is taxi busy?

load (q: INTEGER)
  -- Load q passengers.

position: COORDINATE
  -- Current position on map.

From class:

EVENT_TAXI
  VEHICLE
    TAXI
      EVENT_TAXI
    TRAM
    BUS

MOVING

*
Inheriting features

```ruby
defered class VEHICLE
  inherit MOVING
  feature
    [... Rest of class ...]
  end
end
```

All features of `MOVING` are applicable to instances of `VEHICLE`

```ruby
defered class TAXI
  inherit VEHICLE
  feature
    [... Rest of class ...]
  end
end
```

All features of `VEHICLE` are applicable to instances of `TAXI`

```ruby
class EVENT_TAXI
  inherit TAXI
  feature
    [... Rest of class ...]
  end
end
```

All features of `TAXI` are applicable to instances of `EVENT_TAXI`
Inherited features

\[ m: \text{MOVING}; v: \text{VEHICLE}; t: \text{TAXI}; e: \text{EVENT\_TAXI} \]

\[ v.\text{load}(\ldots) \]
\[ e.\text{take}(\ldots) \]
\[ m.\text{position} \quad \text{-- An expression} \]
\[ t.\text{busy} \quad \text{-- An expression} \]
\[ e.\text{load}(\ldots) \]
\[ e.\text{take}(\ldots) \]
\[ e.\text{position} \quad \text{-- An expression} \]
\[ e.\text{busy} \quad \text{-- An expression} \]
Definitions: kinds of feature

A “feature of a class” is one of:

- An **inherited** feature if it is a feature of one of the parents of the class.

- An **immediate** feature if it is declared in the class, and not inherited. In this case the class is said to **introduce** the feature.
Polymorphic assignment

\( v: \text{VEHICLE} \)
\( \text{cab}: \text{EVENT\_TAXI} \)
\( \text{tram}: \text{TRAM} \)

\( v := \text{cab} \)

More interesting:

\[
\begin{align*}
\text{if } & \text{some\_condition then} \\
& v := \text{cab} \\
\text{else} & \\
& v := \text{tram} \\
\text{end}
\end{align*}
\]
Assignments

Assignment:

\[ \textit{target} := \textit{expression} \]

So far (no polymorphism):

\[ \textit{expression} \text{ was always of the same type as } \textit{target} \]

With polymorphism:

The type of \textit{expression} is a descendent of the type of \textit{target}
Polymorphism is also for argument passing

```hs
register_trip(v: VEHICLE)
  do ... end
```

A particular call:

```hs
register_trip(cab)
```

Type of actual argument is *proper descendant* of type of formal
An **attachment** (assignment or argument passing) is **polymorphic** if its target variable and source expression have different types.

An **entity** or **expression** is **polymorphic** if it may at runtime — as a result of polymorphic attachments — become attached to objects of different types.

**Polymorphism** is the existence of these possibilities.
Definitions (Static and dynamic type)

The **static type** of an entity is the type used in its declaration in the corresponding class text.

If the value of the entity, during a particular execution, is attached to an object, the type of that object is the entity’s **dynamic type** at that time.
v: VEHICLE
cab: EVENT_TAXI

v := cab

Static type of v: VEHICLE

Dynamic type after this assignment: EVENT_TAXI
Basic type property

Static and dynamic type

The dynamic type of an entity will always conform to its static type

(Ensured by the type system)
Static typing

**Type-safe call** (during execution):

A feature call $x.f$ such that the object attached to $x$ has a feature corresponding to $f$

[Generalizes to calls with arguments, $x.f(a, b)$]

**Static type checker:**

A program-processing tool (such as a compiler) that guarantees, for any program it accepts, that any call in any execution will be **type-safe**

**Statically typed language:**

A programming language for which it is possible to write a **static type checker**
Inheritance and static typing

Basic inheritance type rule

For a polymorphic attachment to be valid, the type of the source must conform to the type of the target.

Conformance: basic definition

*Reference* types (non-generic): $U$ *conforms* to $T$ if $U$ is a descendant of $T$

An *expanded* type conforms only to itself.
A reference type $U$ *conforms* to a reference type $T$ if either:

- They have no generic parameters, and $U$ is a descendant of $T$.
- They are both generic derivations with the same number of actual generic parameters, the base class of $U$ is a descendant of the base class of $T$, and every actual parameter of $U$ (recursively) conforms to the corresponding actual parameter of $T$.

An expanded type conforms only to itself.
Static typing (reminder)

**Type-safe call** (during execution):

A feature call $x.f$ such that the object attached to $x$ has a feature corresponding to $f$.

[Generalizes to calls with arguments, $x.f(a, b)$]

**Static type checker:**

A program-processing tool (such as a compiler) that guarantees, for any program it accepts, that any call in any execution will be *type-safe*.

**Statically typed language:**

A programming language for which it is possible to write a *static type checker*. 
Another example hierarchy

- FIGURE
  - center*
  - display*
  - rotate*
  - OPEN_ FIGURE
    - perimeter*
    - SEGMENT
    - POLYLINE
    - TRAINE
    - POLYGON
    - RECTANGLE
    - SQUARE
  - CLOSED_ FIGURE
    - perimeter*
    - ELLIPSE
      - perimeter*
      - CIRCLE
        - perimeter**
Redefinition 1: polygons

class POLYGON inherit CLOSED FIGURE
create
make
feature
  vertex: ARRAY [POINT]
  vertex_count: INTEGER
  perimeter: REAL
      -- Perimeter length.
  do
    from ... until ... loop
      Result := Result + vertex [i] . distance (vertex [i + 1])
    end
  end
invariant
  vertex_count >= 3
  vertex_count = vertex.count
end
Redefinition 2: rectangles

class RECTANGLE inherit
  POLYGON
  redefine perimeter
end

create
  make

feature
  diagonal, side1, side2: REAL
  perimeter: REAL
    -- Perimeter length.
    do Result := 2 * (side1 + side2) end

invariant
  vertex_count = 4
end
Inheritance, typing and polymorphism

Assume:

\[ p: \text{POLYGON} ; \quad r: \text{RECTANGLE} ; \quad t: \text{TRIANGLE} \]
\[ x: \text{REAL} \]

Permitted:

\[ x := p.\text{perimeter} \]
\[ x := r.\text{perimeter} \]
\[ x := r.\text{diagonal} \]
\[ p := r \]

NOT permitted:

\[ x := p.\text{diagonal} \]
\[ r := p \]

-- Even just after \( p := r \)!
Dynamic binding

What is the effect of the following (if `some_test` is true)?

```plaintext
if some_test then
  p := r
else
  p := t
end
x := p.perimeter
```

Redefinition: A class may change an inherited feature, as with `POLYGON` redefining `perimeter`.

Polymorphism: `p` may have different forms at run-time.

Dynamic binding: Effect of `p.perimeter` depends on run-time form of `p`. 
Dynamic binding (a semantic rule):

- Any execution of a feature call will use the version of the feature best adapted to the type of the target object
Binding and typing

(For a call $x \cdot f$)

Static typing: The guarantee that there is at least one version for $f$

Dynamic binding: The guarantee that every call will use the most appropriate version of $f$
Without dynamic binding?

```plaintext
display (f: FIGURE)
  do
    if "f is a CIRCLE" then
      ...
      else if "f is a POLYGON" then
        ...
      end
  end
end
```

and similarly for all other routines!

Tedious; must be changed whenever there's a new figure type
With inheritance and associated techniques

With:

\[
\begin{align*}
  f & : \text{FIGURE} \\
  c & : \text{CIRCLE} \\
  p & : \text{POLYGON}
\end{align*}
\]

Initialize:

\[
\begin{align*}
  \text{if} \, ... & \text{ then} \\
  f & := c \\
  \text{else} & \\
  f & := p \\
  \text{end}
\end{align*}
\]

and:

\[
\begin{align*}
  \text{create} & \, \text{c}\cdot\text{make}(...) \\
  \text{create} & \, \text{p}\cdot\text{make}(...)
\end{align*}
\]

Then just use:

\[
\begin{align*}
  f\cdot\text{move}(...) \\
  f\cdot\text{rotate}(...) \\
  f\cdot\text{display}(...) \\
  \text{-- and so on for every} \\
  \text{-- operation on } f!
\end{align*}
\]
Inheritance: summary 1

Type mechanism: lets you organize our data abstractions into taxonomies

Module mechanism: lets you build new classes as extensions of existing ones

Polymorphism: Flexibility *with* type safety

Dynamic binding: automatic adaptation of operation to target, for more modular software architectures
Redefinition

defered class MOVING feature
  origin: COORDINATE
  destination: COORDINATE
  position: COORDINATE
  polycursor: LIST[COORDINATE]

update_coordinates
  -- Update origin and destination.
  do
    [...]
    origin := destination
    polycursor.forth
    destination := polycursor.item
    [...]
  end
  [...]
end
Redefinition 2: LINE_VEHICLE

deferred class LINE_VEHICLE inherit VEHICLE
    redefine update_coordinates end

feature
    linecursor: LINE_CURSOR
    update_coordinates
        -- Update origin and destination.
        do
            [...]
            origin := destination
            polycursor.forth
            if polycursor.after then
                linecursor.forth
                create polycursor.make (linecursor.item.polypoints)
                polycursor.start
            end
            destination := polycursor.item
        end
end
Dynamic binding

What is the effect of the following (assuming `some_test` true)?

\[
m: MOVING, l: LINE\_VEHICLE, t: TAXI
\]

\[
\text{if } some\_test \text{ then}
\]
\[
\quad m := l
\]
\[
\text{else}
\]
\[
\quad m := t
\]
\[
\text{end}
\]

\[
m.\text{update\_coordinates}
\]

Redefinition: A class may change an inherited feature, as with `LINE\_VEHICLE` redefining `update\_coordinates`.

Polymorphism: \(m\) may have different forms at run-time.

Dynamic binding: Effect of \(m.\text{update\_coordinates}\) depends on run-time form of \(m\).
There are multiple versions of `take`. 
Extending the basic notion of class

- Abstraction
- Inheritance
- Genericity
- Type parameterization

LIST_OF_CARS

SET_OF_CARS

LIST_OF_CITIES

LIST_OF_PERSONS

LINKED_LIST_OF_CARS
Extending the basic notion of class
Conformance

Defined earlier for non-generically derived types:
Polymorphic data structures

\[ \text{fleet: LIST [VEHICLE]} \]
\[ \text{v: VEHICLE} \]

\[ \text{extend(v: G)} \]

\[ \text{-- Add a new occurrence of } v. \]

\[ \text{...} \]
\[ \text{fleet.extend(v)} \]
\[ \text{fleet.extend(cab)} \]
Definition (Polymorphism, adapted)

An attachment (assignment or argument passing) is polymorphic if its target entity and source expression have different types.

An entity or expression is polymorphic if - as a result of polymorphic attachments - it may at runtime become attached to objects of different types.

A container data structure is polymorphic if it may contain references to objects of different types.

Polymorphism is the existence of these possibilities.
What we have seen

The basics of fundamental O-O mechanisms:

- Inheritance
- Polymorphism
- Dynamic binding
- Static typing
- Genericity
Our program for the second part

Reminder on genericity, including constrained

Inheritance: deferred classes
Inheritance: what happens to contracts?

Inheritance: how do we find the actual type of an object?

Still to see about inheritance after this lecture: multiple inheritance, and various games such as renaming
Genericity (reminder)

Unconstrained

\[ \text{LIST}[G] \]

e.g. \( \text{LIST}[	ext{INTEGER}] \), \( \text{LIST}[	ext{PERSON}] \)

Constrained

\[ \text{HASH\_TABLE}[G \rightarrow \text{HASHABLE}] \]
\[ \text{VECTOR}[G \rightarrow \text{NUMERIC}] \]
A generic class (reminder)

class $LIST[G]$ feature
    extend ($x: G$) ...
    last: $G$ ...
end

To use the class: obtain a \textit{generic derivation}, e.g.

$\text{cities}: LIST[CITY]$
Using generic derivations (reminder)

```plaintext
cities : LIST[CITY]
people : LIST[PERSON]
c : CITY
p : PERSON
...
cities.extend (c)
people.extend (p)
c := cities.last
c.some_city_operation
```

**STATIC TYPING**
The compiler will reject:

- `people.extend (c)`
- `cities.extend (p)`
Genericty: summary 1

- Type extension mechanism
- Reconciles flexibility with type safety
- Enables us to have parameterized classes
- Useful for container data structures: lists, arrays, trees, ...
- “Type” now a bit more general than “class”
Definition: Type

We use types to declare entities, as in

\[ x: SOME\_TYPE \]

With the mechanisms defined so far, a type is one of:

- A non-generic class
  
  e.g. \[ METRO\_STATION \]

- A generic derivation, i.e. the name of a class followed by a list of types, the actual generic parameters, in brackets
  
  e.g. \[ LIST[\{METRO\_STATION\}] \]
  \[ LIST[ARRAY[\{METRO\_STATION\}]] \]
Combining genericity with inheritance

- Abstraction
- Inheritance
- Genericity
- Specialization

Type parameterization

LIST_OF_CITIES

LIST_OF_CARS

SET_OF_CARS

LINKED_LIST_OF_CARS

LIST_OF_PERSONS
Genericity + inheritance 1: Constrained genericity

```plaintext
class VECTOR [G] feature
  plus alias "+" (other: VECTOR [G]): VECTOR [G]
    -- Sum of current vector and other.
    require
      lower = other.lower
      upper = other.upper
  local
    a, b, c: G
  do
    ... See next ...
  end
  ... Other features ...
end
```
Adding two vectors

\[ u + v = w \]
Constrained genericity

Body of \textit{plus alias} 
\texttt{"\texttt{+}"}:

\begin{verbatim}
create Result.make(lower, upper)

from
i := lower
until
i > upper
loop
a := item(i)
b := other.item(i)
c := a + b  -- Requires \texttt{"\texttt{+}"} operation on \textit{G}!
Result.put(c, i)
i := i + 1
end
\end{verbatim}
The solution

Declare class \texttt{VECTOR} as

\begin{verbatim}
  class \texttt{VECTOR} [G \rightarrow \texttt{NUMERIC}] feature
    ... The rest as before ...
  end
\end{verbatim}

Class \texttt{NUMERIC} (from the Kernel Library) provides features \texttt{plus alias "+", minus alias "-"} and so on.
Improving the solution

Make \texttt{VECTOR} itself a descendant of \texttt{NUMERIC}, effecting the corresponding features:

\begin{verbatim}
class VECTOR [ G -> NUMERIC ] inherit NUMERIC

feature

... Rest as before, including \texttt{infix "+"}...

end
\end{verbatim}

Then it is possible to define

\begin{verbatim}
v : VECTOR [ INTEGER ]
vv : VECTOR [ VECTOR [ INTEGER ] ]
vvv : VECTOR [ VECTOR [ VECTOR [ INTEGER ] ] ]
\end{verbatim}
Combining genericity with inheritance

**Type parameterization**
- LIST_OF_CITIES
- LIST_OF_CARS
- LINKED_LIST_OF_CARS

**Genericity**
- LIST_OF_CARS
- LIST_OF_CITIES
- LIST_OF_PERSONS

**Abstraction**
- SET_OF_CARS

**Inheritance**
- Specialization

**Type parameterization**
Genericity + inheritance 2: Polymorphic data structures

```
figs : LIST [FIGURE]
p1, p2 : POLYGON
c1, c2 : CIRCLE
e : ELLIPSE

class LIST[G] feature
    extend(v : G) do ...
    last : G
    ...
end

figs.extend(p1) ; figs.extend(c1) ; figs.extend(c2)
figs.extend(e) ; figs.extend(p2)
```

(POLYGON) (CIRCLE) (CIRCLE) (ELLIPSE) (POLYGON)
Example hierarchy

- FIGURE
  - * OPEN FIGURE
    - SEGMENT
    - POLYLINE
  - CLOSED FIGURE
    - POLYGON
      - TRIANGLE
      - SQUARE
      - CIRCLE
  - ELLIPSE

- center*
- display*
- rotate*
- perimeter *
- perimeter +
- perimeter **
- side1
- side2
- diagonal
- deferred
- effective
- + redefine
- perimeter **
- perimeter **
Another application: undoing-redoing

This example again uses a powerful polymorphic data structure
This will only be a sketch; we’ll come back to the details in the agent lecture

References:

- Chapter 21 of my *Object-Oriented Software Construction*, Prentice Hall, 1997
- Erich Gamma et al., *Design Patterns*, Addison-Wesley, 1995: “Command pattern”
Enabling users of an interactive system to cancel the effect of the last command

Often implemented as “Control-Z”

Should support multi-level undo-redo (“Control-Y”), with no limitation other than a possible maximum set by the user
Our working example: a text editor

Notion of “current line”.
Assume commands such as:

- **Remove** current line
- **Replace** current line by specified text
- **Insert** line before current position
- **Swap** current line with next if any
- “Global search and replace” (hereafter *GSR*): replace every occurrence of a specified string by another
- ...

This is a line-oriented view for simplicity, but the discussion applies to more sophisticated views.
A straightforward solution

Before performing any operation, save entire state

In the example: text being edited, current position in text

If user issues "Undo" request, restore entire state as last saved

But: huge waste of resources, space in particular

Intuition: only save the "diff" between states.
Keeping the history of the session

The history list:

- Insertion
- Insertion
- Removal
- Insertion
- Swap

Oldest -> Most recent

History: TWO WAY LIST [COMMAND]
What’s a “command” object?

A command object includes information about one execution of a command by the user, sufficient to:

- Execute the command
- Cancel the command if requested later

For example, in a **Removal** command object, we need:

- The position of the line being removed
- The content of that line!
General notion of command

defered class COMMAND feature

  done: BOOLEAN
    -- Has this command been executed?

eexecute
    -- Carry out one execution of this command.

defered
  ensure
    already: done

end

undo
  -- Cancel an earlier execution of this command.

require
  already: done

defered
end
end
Command class hierarchy

- **COMMAND**
  - execute*: deferred
  - undo*: effective

  **REMOVAL**
  - execute*
  - undo*
  - line: STRING
  - index: INTEGER
  - ...

  **INSERTION**
  - execute*
  - undo*
  - index
  - ...

  ...
class EDIT_CONTROLLER feature

text : TWO_WAY_LIST[STRING]

remove

require not off

do text . remove
end

put_right (line : STRING)

require not after

do text . put_right (line)
end

... also item, index, go_ith, put_left ...
end
class REMOVAL inherit COMMAND feature

controller: EDIT_CONTROLLER
    -- Access to business model.

line: STRING
    -- Line being removed.

index: INTEGER
    -- Position of line being removed.

execute
    -- Remove current line and remember it.
    do
      line := controller.item; index := controller.index
      controller.remove ; done := True
    end

undo
    -- Re-insert previously removed line.
    do
      controller.go_i_th(index)
      controller.put_left(line)
    end
end
The history list

A polymorphic data structure:

history: TWO_WAY_LIST [COMMAND]
Reminder: the list of figures

class

\[ \text{LIST}[G] \]

feature

... \[ \text{last: } G \text{ do } ... \]

extend (x: G) do ... \n
end

\[ fl: \text{LIST}[\text{FIGURE}] \]
\[ r: \text{RECTANGLE} \]
\[ s: \text{SQUARE} \]
\[ t: \text{TRIANGLE} \]
\[ p: \text{POLYGON} \]

...

\[ fl.\text{extend}(p); \ fl.\text{extend}(t); \ fl.\text{extend}(s); \ fl.\text{extend}(r) \]

\[ fl.\text{last}.\text{display} \]
Reminder: the list of figures

\[
\text{figs.extend}(p1) ; \text{figs.extend}(c1) ; \text{figs.extend}(c2) \\
\text{figs.extend}(e) ; \text{figs.extend}(p2)
\]

\[
\text{figs}: \text{LIST}[\text{FIGURE}] \\
p1, p2: \text{POLYGON} \\
c1, c2: \text{CIRCLE} \\
e: \text{ELLIPSE}
\]

\[
\text{class } \text{LIST}[G] \text{ feature} \\
\text{extend}(v: G) \text{ do } \ldots \\
\text{end} \\
\text{last}: G \\
\ldots \\
\text{end}
\]
The history list

A polymorphic data structure:

```
history: TWO_WAY_LIST [COMMAND]
```

![Diagram showing history list with operations: Insertion, Insertion, Removal, Insertion, Swap, Oldest to Most recent]

Executing a user command

decode_user_request

if "Request is normal command" then
   "Create command object \( c \) corresponding to user request"
   history.extend(\( c \))
   \( c \).execute

elseif "Request is UNDO" then
   if not history.before then
      -- Ignore excessive requests
      history.item.undo
      history.back
   end

elseif "Request is REDO" then
   if not history.is_last then
      -- Ignore excessive requests
      history.forth
      history.item.execute
   end
end
Command class hierarchy

execute*
undo*

* deferred
+ effective

execute*
undo*
line: STRING
index: INTEGER

execute*
undo*
index

...
Example hierarchy

```
* deferred
+ effective
++ redefined

FIGURE

* center*
* display*
* rotate*

OPEN_ FIGURE

* perimeter*

POLYLINE

SEGMENT

POLYGON

RECTANGLE

SQUARE

ELLIPSE

* perimeter+

TRIANGLE

CIRCLE

* perimeter+

+ perimeter+

side1 side2 diagonal

perimeter++

perimeter++

perimeter++

...
Enforcing a type: the problem

\[ \text{fl.store(“FN”)} \]

... 

\[ \text{-- Two years later:} \]
\[ \text{fl := retrieved(“FN”)} \]
\[ \text{-- See next} \]
\[ x := \text{fl.last} \]
\[ \text{-- [1]} \]
\[ \text{print(x.diagonal)} \]
\[ \text{-- [2]} \]

What’s wrong with this?

- If \[ x \] is declared of type \textit{RECTANGLE}, [1] is invalid.
- If \[ x \] is declared of type \textit{FIGURE}, [2] is invalid.
Enforcing a type: the Object Test

if \texttt{attached \{RECTANGLE\} \texttt{fl.retrieved("FN") as r}} then

\texttt{print (r.diagonal)}

... Do anything else with \texttt{r}, guaranteed

... to be non void and of dynamic type \texttt{RECTANGLE}

else

\texttt{print("Too bad.")}

end

SCOPE of the Object-Test Local
Earlier mechanism: assignment attempt

\[ f: \text{FIGURE} \]
\[ r: \text{RECTANGLE} \]

... 
\[ fl.\text{retrieve} ("FN") \]
\[ f := fl.\text{last} \]

\[ r? = f \]

if \( r \neq \text{Void} \) then
  \[ \text{print} (r.\text{diagonal}) \]
else
  \[ \text{print} ("Too bad.") \]
end
Assignment attempt

$x?= y$

with

$x: A$

Semantics:

- If $y$ is attached to an object whose type conforms to $A$, perform normal reference assignment.
- Otherwise, make $x$ void.
The role of deferred classes

Express abstract concepts independently of implementation

Express common elements of various implementations

Terminology: **Effective** = non-deferred
(i.e. fully implemented)
A deferred feature

In e.g. \textit{LIST}:

\begin{verbatim}
forth
require not after
deferred
ensure \textit{index} = old \textit{index}
end
\end{verbatim}
Mixing deferred and effective features

In the same class

Effective!

search(x: G)

-- Move to first position after current
-- where x appears, or after if none.

do
from until after or else item = x loop

forth

end
end

Deferred!

“Programs with holes”
“Don’t call us, we’ll call you!”

A powerful form of reuse:

- The reusable element defines a general scheme
- Specific cases fill in the holes in that scheme

Combine reuse with adaptation
Applications of deferred classes

Analysis and design, top-down

Taxonomy

Capturing common behaviors
Deferred classes in EiffelBase
Java and .NET solution

Single inheritance only for classes
Multiple inheritance from interfaces

An interface is like a fully deferred class, with no implementations (do clauses), no attributes (and also no contracts)
Multiple inheritance: Combining abstractions

COMPARABLE

<, <=, >, >=, ...

(total order relation)

NUMERIC

+ , − , * , /...

(commutative ring)

INTEGER

REAL

STRING

COMPLEX
How do we write `COMPARABLE`?

defered class `COMPARABLE` feature

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

    less_equal alias "<=" (x: COMPARABLE): BOOLEAN
    do Result := (Current < x or (Current = x))
    end

    greater alias ">" (x: COMPARABLE): BOOLEAN
    do Result := (x < Current) end

    greater_equal alias ">=" (x: COMPARABLE): BOOLEAN
    do Result := (x <= Current) end

Deferred classes vs Java interfaces

Interfaces are “entirely deferred”:
  Deferred features only

Deferred classes can include effective features, which rely on deferred ones, as in the *COMPARABLE* example

Flexible mechanism to implement abstractions progressively
Applications of deferred classes

Abstraction

Taxonomy

High-level analysis and design

...
class \textit{SCHEDULE} feature

\hspace{2em} \textit{segments}: \textit{LIST[SEGMENT]}

end

\textit{Source: Object-Oriented Software Construction, 2\textsuperscript{nd} edition, Prentice Hall}
Schedules

**note**

*description*:  
"24-hour TV schedules"

defered class SCHEDULE feature

segments : LIST [SEGMENT]  
--- Successive segments.

defered
end

air_time : DATE  
--- 24-hour period  
--- for this schedule.

defered
end

**set_air_time (t : DATE)**

--- Assign schedule to
--- be broadcast at time t.

require

t.in_future

defered
ensure

air_time = t
end

print

--- Produce paper version.

defered
end
end
Segment

**note**

`description` : "Individual fragments of a schedule"

**deferred class SEGMENT feature**

`schedule : SCHEDULE deferred end`

-- Schedule to which
-- segment belongs.

`index : INTEGER deferred end`

-- Position of segment in
-- its schedule.

`starting_time, ending_time :`

`INTEGER deferred end`

-- Beginning and end of
-- scheduled air time.

`next : SEGMENT deferred end`

-- Segment to be played
-- next, if any.

**sponsor : COMPANY deferred end**

-- Segment's principal sponsor.

**rating : INTEGER deferred end**

-- Segment's rating (for
-- children's viewing etc.).

... Commands such as
`change_next, set_sponsor,
set_rating`, omitted ...

**Minimum_duration : INTEGER = 30**

-- Minimum length of segments,
-- in seconds.

**Maximum_interval : INTEGER = 2**

-- Maximum time between two
-- successive segments, in seconds.
Segment (continued)

**invariant**

\[ \text{in\_list: } (1 \leq index) \text{ and } (index \leq schedule\_segments\_count) \]

\[ \text{in\_schedule: } schedule\_segments\_item(index) = \text{Current} \]

\[ \text{next\_in\_list: } (next \neq \text{Void}) \text{ implies } \]
\[ (schedule\_segments\_item(index + 1) = next) \]

\[ \text{no\_next\_iff\_last: } (next = \text{Void}) = (index = schedule\_segments\_count) \]

\[ \text{non\_negative\_rating: } rating \geq 0 \]

\[ \text{positive\_times: } (starting\_time \geq 0) \text{ and } (ending\_time \geq 0) \]

\[ \text{sufficient\_duration: } \]
\[ ending\_time - starting\_time \geq \text{Minimum\_duration} \]

\[ \text{decent\_interval: } \]
\[ (next.\text{starting\_time}) - ending\_time \leq \text{Maximum\_interval} \]

end
Commercial

```plaintext
note
description: "Advertizing segment"
defered class COMMERCIAL
    inherit SEGMENT
    rename sponsor as advertizer end

feature
    primary: PROGRAM deferred
        -- Program to which this
        -- commercial is attached.
    primary_index: INTEGER deferred
        -- Index of primary.

set_primary(p: PROGRAM)
    -- Attach commercial to p.
require
    program_exists: p /= Void
    same_schedule:
        p.schedule = schedule
    before:
        p.starting_time <= starting_time
defered
ensure
    index_updated:
        primary_index = p.index
    primary_updated: primary = p
end
```
 Commercial (continued)

\textbf{invariant}
\begin{align*}
\text{meaningful\_primary\_index}: & \quad \text{primary\_index} = \text{primary\_.index} \\
\text{primary\_before}: & \quad \text{primary\_.starting\_time} \leq \text{starting\_time} \\
\text{acceptable\_sponsor}: & \quad \text{advertizer\_.compatible (primary\_.sponsor)} \\
\text{acceptable\_rating}: & \quad \text{rating} \leq \text{primary\_.rating}
\end{align*}
\textbf{end}
deferred class
  VAT
inherit
  TANK
feature
  in_valve, out_valve: VALVE
    -- Fill the vat.
    require
      in_valve.open
      out_valve.closed
    deferred
    ensure
      in_valve.closed
      out_valve.closed
      is_full
  end

  empty, is_full, is_empty, gauge, maximum, ... [Other features] ...

invariant
  is_full = (gauge >= 0.97 * maximum) and (gauge <= 1.03 * maximum)
end
Contracts and inheritance

Issue: 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 in $C$:

\[
\text{if } a1.\alpha \text{ then } a1.r(...) \quad \text{-- Here } a1.\beta \text{ holds}
\]

end
Assertion redeclaration rule

When redeclaring a routine, we may only:

- Keep or weaken the precondition
- Keep or strengthen the postcondition
A simple language rule does the trick!

Redefined version may have nothing (assertions kept by default), or

```
require else new_pre
ensure then new_post
```

Resulting assertions are:

- `original_precondition or new_pre`
- `original_postcondition and new_post`
What we have seen

Deferred classes and their role in software analysis and design

Contracts and inheritance

Finding out the “real” type of an object