Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

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
But first…

Some complements about the object model
Variants of assignment and copy

Reference assignment ($a$ and $b$ of reference types):

\[ b := a \]

Object duplication (shallow):

\[ c := a.twin \]

Object duplication (deep):

\[ d := a.deep_twin \]

Also: shallow field-by-field copy (no new object is created):

\[ e.copy(a) \]
Shallow and deep cloning

Initial situation:

Result of:

\[ b := a \]

\[ c := a.twin \]

\[ d := a.deep_twin \]
Where do these mechanisms come from?

Class **ANY** in the Eiffel “Kernel Library”

Every class that doesn’t explicitly inherit from another is considered to inherit from **ANY**

As a result, every class is a descendant of **ANY**.
Completing the inheritance structure

A

B

C

D

E

ANY

NONE

Inherits from
A related mechanism: Persistence

\[ a.\text{store}(\text{file}) \]

\[ \ldots \]

\[ b := \text{retrieved}(\text{file}) \]

Storage is automatic.
Persistent objects identified individually by keys.

These features come from the library class \textit{STORABLE}.
and now…

…to inheritance and genericity
Extending the basic notion of class

Abstraction

Inheritance

Genericity

Type parameterization

Specialization
Extending the basic notion of class
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?

Dubious use of a container data structure:

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

What if wrong?

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

\[
\begin{align*}
  c &= \text{cities.last} \\
  c &: \text{some\_city\_operation}
\end{align*}
\]
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, now also found in Java, .NET and others.
A generic class

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

To use the class: obtain a generic derivation, e.g.

cities: LIST[ CITY ]
```
Using generic derivations

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**

**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

\texttt{LIST [CITY]}
\texttt{LIST [LIST [CITY]]}

\ldots

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

(But every type remains \textbf{based} on a class.)
A generic class

class \textit{LIST}[G] \textit{feature}

\begin{itemize}
  \item \textit{extend} (x: G) ...
  \item \textit{last}: G ...
\end{itemize}

\textbf{end}

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

cities: \textit{LIST}[\textit{CITY}]
Extending the basic notion of class

- Abstraction
- Inheritance
- Genericity
- Type parameterization
- Specialization
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*)
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

Parent, Heir

Ancestor, Descendant
The ancestors of \( B \) are \( B \) itself and the ancestors of its parents.

Proper ancestor, Proper descendant

Direct instance, Instance
The instances of \( A \) are the direct instances of its descendants.

(Other terminology: subclass, superclass, base class)
If $B$ inherits from $A$ (by listing $A$ in its inherit clause), $B$ is an heir of $A$ and $A$ is a parent of $B$.

The descendants of a class are the class itself and (recursively) the descendants of its heirs; proper descendants exclude the class itself.

“Ancestor” and “proper ancestor” are the reverse notions.
Example hierarchy (from Traffic)

* Deferred
+ Effective
++ Redefined

MOVING

position
update_coordinates
move

VEHICLE
load

* TAXI
busytake*

* LINE_VEHICLE
update_coordinates**
move**

take+

EVENT_TAXI

TRAM

BUS
Features in the example

**Feature**

\[\text{take (from}\_\text{location, to}\_\text{location}: \text{COORDINATE})\]

-- Bring passengers from *from_location*
-- to *to_location*.

\[\text{busy: BOOLEAN}\]
-- Is taxi busy?

\[\text{load (q: INTEGER)}\]
-- Load *q* passengers.

\[\text{position: TRAFFIC\_COORDINATE}\]
-- Current position on map

**From class:**

- \text{EVENT\_TAXI}
- \text{TAXI}
- \text{VEHICLE}
- \text{MOVING}
Inheriting features

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

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

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

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

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

All features of `TAXI` are applicable to instances of `EVENT_TAXI`.
Example hierarchy

* deferred
+ effective
++ redefined

MOVING

VEHICLE

TAXI

EVENT_TAXI

LINE_VEHICLE

TRAM

BUS

position
update_coordinates
move
load

update_coordinates++
move++

busy
take*
take+

* deferred
+ effective
++ redefined
Inherited features

\[ m: MOVING, v: VEHICLE, t: TAXI, e: EVENT\_TAXI \]

\[
\begin{align*}
&v \cdot load (...) \\
&e \cdot take (...) \\
&m \cdot position &\text{-- An expression} \\
&t \cdot busy &\text{-- An expression} \\
&e \cdot load (...) \\
&e \cdot take (...) \\
&e \cdot position &\text{-- An expression} \\
&e \cdot busy &\text{-- An expression}
\end{align*}
\]
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

\[ v : \text{VEHICLE} \]
\[ \text{cab} : \text{EVENT\_TAXI} \]

\[ v := \text{cab} \]
Assignments

Assignment:

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

So far (no polymorphism):

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

With polymorphism:

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

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

A particular call:

```
register_trip(cab)
```

Type of actual argument is *proper descendant* of type of formal
Definitions: Polymorphism

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.
Static and dynamic type

\[ v: \text{VEHICLE} \]
\[ \text{cab : EVENT\_TAXI} \]

\[ v := \text{cab} \]

Static type of \( v \): \text{VEHICLE}

Dynamic type after this assignment: \text{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): $U \text{ conforms to } T$ if $U$ is a descendant of $T$

An expanded type conforms only to itself
Conformance: full definition

A reference type $U$ \textbf{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

* deferred
+ effective
++ redefined
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:

\[ p: POLYGON \ ; \ r: RECTANGLE \ ; \ t: TRIANGLE \]
\[ x: REAL \]

Permitted:

\[ x := p\.perimeter \]
\[ x := r\.perimeter \]
\[ x := r\.diagonal \]
\[ p := r \]

NOT permitted:

\[ x := p\.diagonal \quad -- \text{Even just after } p := r ! \]
\[ r := p \]
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`. 
Without dynamic binding?

\[
\text{display } (f: \text{FIGURE})
\]
\[
\text{do}
\]
\[
\text{if } "f \text{ is a CIRCLE}" \text{ then}
\]
\[
\text{...}
\]
\[
\text{elseif } "f \text{ is a POLYGON}" \text{ then}
\]
\[
\text{...}
\]
\[
\text{end}
\]
\[
\text{end}
\]

and similarly for all other routines!

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

With:

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

Initialize:

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

and:

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

Then just use:

\[
\begin{align*}
\text{f.move(...)} \\
\text{f.rotate(...)} \\
\text{f.display(...)} \\
\quad \text{-- and so on for every} \\
\quad \text{-- operation on } f
\end{align*}
\]
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

deferred 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

[Diagram showing movement of polycursor and origin and destination points]
Redefinition 2: LINE_VEHICLE

defered 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
            polycursor.forth
        end

    end
Dynamic binding

What is the effect of the following (assuming \textit{some\_test} true)?
\begin{verbatim}
m: MOVING, l: LINE\_VEHICLE, t: TAXI

    if some\_test then
      m := l
    else
      m := t
    end

m.update\_coordinates
\end{verbatim}

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

Polymorphism: \textit{m} may have different forms at run-time.

Dynamic binding: Effect of \textit{m.update\_coordinates} depends on run-time form of \textit{m}.
Dynamic binding

There are multiple versions of \textit{take}.

\begin{center}
\begin{tikzpicture}
  \node [circle, draw=green!50!black, fill=green!20] (T1) at (0,0) {
    \begin{tabular}{c}
      \textit{*} \vspace{0.5em} \\
      \textsc{TAXI} \vspace{0.5em} \\
    \end{tabular}
  };

  \node [circle, draw=green!50!black, fill=green!20] (T2) at (-2,-2) {
    \begin{tabular}{c}
      \textit{take} \vspace{0.5em} \\
      \textsc{EVENT\_TAXI} \vspace{0.5em} \\
    \end{tabular}
  };

  \node [circle, draw=green!50!black, fill=green!20] (T3) at (2,-2) {
    \begin{tabular}{c}
      \textit{take} \vspace{0.5em} \vspace{0.5em} \\
      \textsc{DISPATCHER\_TAXI} \vspace{0.5em} \\
    \end{tabular}
  };

  \draw [->, red] (T1) to (T2);
  \draw [->, red] (T1) to (T3);

  \node [below of = T1] (bus) {\textit{busy} \vspace{0.5em} \textit{take*}};

  \node [above of = T1, yshift=1cm] (inher) \text{Inherits from} \vspace{0.5em} \text{* Deferred};
\end{tikzpicture}
\end{center}
**Definitions (Dynamic binding)**

*Dynamic binding* (a semantic rule) is the property that any execution of a feature call will use the version of the feature best adapted to the type of the target object.
Extending the basic notion of class

Abstraction

Inheritance

Genericity

Type parameterization

Specialization
Extending the basic notion of class

Inheritance

Genericity
Conformance

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

\( fleet: LIST [VEHICLE] \)
\( v: VEHICLE \)

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

-- Add a new occurrence of \( v \).

... 

\( fleet.\text{extend}(v) \)
\( fleet.\text{extend}(\text{cab}) \)
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

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

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

Constrained

\[ \text{HASH}_\text{TABLE}[G \rightarrow \text{HASHABLE}] \]

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

class \text{LIST}\[G\] feature
  \text{extend}(x : G) \ldots
  \text{last} : G \ldots
end

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

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

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)
Genericity: 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

\[ x: \text{SOME\_TYPE} \]

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

- A non-generic class
  e.g. \text{METRO\_STATION}

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

**Genericity**
- Type parameterization
- Specialization

**Inheritance**
- Abstraction

**Type**
- LIST_OF_CARS
- LINKED_LIST_OF_CARS
- LIST_OF_CITIES
- LIST_OF_PERSONS

**Set of Cars**
- SET_OF_CARS
Genericity + inheritance 1: Constrained genericity

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

\[ U + V = W \]

2

1

\[ a + b = c \]
Constrained genericity

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

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

Declare class \textit{VECTOR} as

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

Class \textit{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 \rightarrow 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

- Abstraction
- Inheritance
- Genericity
- Specialization
- Type parameterization

- LIST_OF_CARS
- LIST_OF_CITIES
- SET_OF_CARS
- LIST_OF_CARS
- LINKED_LIST_OF_CARS
- LIST_OF_PERSONS
Genericity + inheritance 2: Polymorphic data structures

```plaintext
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)
```

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

* deferred
+ effective
++ redefined
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”
The problem

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:

\[ \text{history: \textsc{TWO\_WAY\_LIST [COMMAND]} } \]
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!
deferred class  COMMAND  feature

done: BOOLEAN
   -- Has this command been executed?

execute
   -- 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**
- **undo**

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

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

**deferr**
- **ed effective**

...
Underlying class (from business model)

class EDIT_CONTROLLER feature

text: TWO WAY LIST [STRING]
remove
  -- Remove line at current position.
  require not off
do
  text.remove
end
put_right (line: STRING)
  -- Insert line after current position.
  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
The history list

A polymorphic data structure:

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

class
   LIST[G]

feature
   ...
   last: G do ...
   extend (x: G) do ...
   end

fl: LIST[FIGURE]
r: RECTANGLE
s: SQUARE
t: TRIANGLE
p: POLYGON
...
fl.extend (p); fl.extend (t); fl.extend (s); fl.extend (r)
fl.last.display
Reminder: the list of figures

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

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

\[
\text{class LIST[G] feature} \\
\text{extend}(v: G) \text{ do } ... \\
\text{last: G} \\
\text{ ... } \\
\text{end}
\]

(POLYGON) (CIRCLE) (CIRCLE) (ELLIPSE) (POLYGON)
The history list

A polymorphic data structure:

\[
\text{history: TWO\_WAY\_LIST [COMMAND]}
\]
Executing a user command

decode_user_request

if "Request is normal command" then
    "Create command object \( c \) corresponding to user request"
    \[
    \text{history.extend}(c) \\
    c.execute
    \]
elseif "Request is UNDO" then
    if not \( \text{history.before} \) then
        \[
        \text{history.item.undo} \\
        \text{history.back}
        \]
    end
elseif "Request is REDO" then
    if not \( \text{history.is_last} \) then
        \[
        \text{history.forth} \\
        \text{history.item.execute}
        \]
    end
end
**Command class hierarchy**

- **COMMAND**
  - execute*
  - undo*

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

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

- **deferr**
  - ed
effecti
  - ve

...
Example hierarchy

- FIGURE
  - OPEN FIGURE
  - CLOSED FIGURE
    - POLYGON
      - TRIANGLE
    - ELLIPSE
      - RECTANGLE
        - SQUARE
  - POLYLINE
  - SEGMENT

* deferred
+ effective
++ redefined

center* rotate*
display*

perimeter*

perimeter+

perimeter++

side1 side2 diagonal
diagonal...
Enforcing a type: the problem

\[ \text{fl.store("FILE\_NAME")} \]

... 

\[ \text{-- Two years later: fl := retrieved("FILE\_NAME") -- See next} \]
\[ \text{x := fl.last -- [1]} \]
\[ \text{print(x.diagomal) -- [2]} \]

What's wrong with this?

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

if \{r: \text{RECTANGLE}\} \text{fl.retrieved("FILE\_NAME")} then

\text{print (r\.diagonal)\n
... Do anything else with } r, \text{ guaranteed\n
... to be non void and of dynamic type \text{RECTANGLE}}

else

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

d\text{end

\text{SCOPE of the Object-Test Local}
Earlier mechanism: assignment attempt

\[ f: \text{FIGURE} \]
\[ r: \text{RECTANGLE} \]
...
\[ fl.\text{retrieve("FILE\_NAME")} \]
\[ f := fl.\text{last} \]

\[ r? = f \]

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

With

\[ x := y \]

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. *LIST*:

```
forth
require
not after
deferred
ensure
index = old index + 1
end
```
Mixing deferred and effective features

In the same class

**search**(*x*: *G*)

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

do

done until **after** or else *item* = *x* loop

**forth**

end end

“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 (\texttt{do} clauses), no attributes (and also no contracts)
Applications of deferred classes

Abstraction

Taxonomy

High-level analysis and design

...
class SCHEDULE feature
    segments : LIST [SEGMENT]
end

Source: Object-Oriented Software Construction, 2nd edition, Prentice Hall
Schedules

note

description : "24-hour TV schedules"

defered class SCHEDULE feature

  segments : LIST [SEGMENT]
    -- Successive segments
      deferred
    end

  air_time : DATE
    -- 24-hour period
    -- for this schedule
      deferred
    end

set_air_time (t : DATE)
  -- Assign schedule to
  -- be broadcast at time t.
  require
    t.in_future
  deferred
  ensure
    air_time = t
  end

print
  -- Produce paper version.
  deferred
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**

- **in_list**: $(1 \leq index)$ and $(index \leq schedule.segments.count)$
- **in_schedule**: $schedule.segments.item(index) = Current$
- **next_in_list**: $(next \neq Void)$ implies $(schedule.segments.item(index + 1) = next)$
- **no_next_iff_last**: $(next = Void) = (index = schedule.segments.count)$
- **non_negative_rating**: $rating \geq 0$
- **positive_times**: $(starting_time > 0)$ and $(ending_time > 0)$
- **sufficient_duration**: $ending_time - starting_time \geq Minimum_duration$
- **decent_interval**: $(next.starting_time) - ending_time \leq Maximum_interval$
note

description:
  "Advertizing segment"

defferred class COMMERCIAL

inhibit
  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
invariant

meaningful_primary_index: primary_index = primary.index
primary_before: primary.starting_time <= starting_time
acceptable_sponsor: advertizer.compatible (primary.sponsor)
acceptable_rating: rating <= primary.rating

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
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?
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 $C$:

```plaintext
if $a1.\alpha$ then
    $a1.r(...)$
    -- Here $a1.\beta$ holds
end
```
Assertion redeclaration rule

When redeclaring a routine, we may only:

- Keep or weaken the precondition
- Keep or strengthen the postcondition
Assertion redeclaration rule in Eiffel

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