Genericity & Inheritance

Bertrand Meyer
ETH Zurich, Switzerland
Eiffel Software, Santa Barbara

Epita, Paris, 31 March 2010
On the menu

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**
- **Specialization**

```
LIST_OF_CARS
SET_OF_CARS
LINKED_LIST_OF_CARS
LIST_OF_CITIES
LIST_OF_PERSONS
```
Extending the basic notion of class
Genericity

Unconstrained

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

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

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
CITY, PERSON
cities : LIST ...
people : LIST ...

people.extend (p)
cities.extend (c)
c := cities.last

c.some_city_operation
```

What if wrong?
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 \texttt{LIST[G]} feature
  extend (x: G) ...
  last: G ...
end

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

cities: \texttt{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
The compiler will reject:
  ➢ people.extend (c)
  ➢ cities.extend (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

\texttt{LIST \{CITY\}}
\texttt{LIST \{LIST \{CITY\}\}}

... 

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

(But every type remains \texttt{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 LIST[\( G \)] feature

   extend (x : G) ... 

   last : G ... 

end

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

cities : LIST[\( \text{CITY} \)]
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
- Inheritance
- Genericity
- Type parameterization
- Specialization

LIST_OF_CARS
LIST_OF_CITIES
LINKED_LIST_OF_CARS
SET_OF_CARS
LIST_OF_PERSONS

Type parameterization

16
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 \textbf{heir of} \( A \)
- \( A \) is a \textbf{parent of} \( B \)

For a class \( A \):

- The \textbf{descendants} of \( A \) are \( A \) itself and (recursively) the descendants of \( A \)'s heirs
- \textbf{Proper descendants} exclude \( A \) itself

Reverse notions:

- Ancestor
- \textbf{Proper ancestor}

More precise notion of instance:

- \textbf{Direct instances} of \( A \)
- \textbf{Instances} of \( A \): the direct instances of \( A \) and its descendants

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

* Deferred
+ Effective
++ Redefined

MOVING

* position
update_coordinates
move

VEHICLE

load

TAXI

* busy
take*
EVENT_TAXI
take+

LINE_VEHICLE

update_coordinates++
move++

TRAM

BUS
# Features in the example

<table>
<thead>
<tr>
<th>Feature</th>
<th>From class:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>take (from_location, to_location: COORDINATE)</code> -- Bring passengers -- from <code>from_location</code> -- to <code>to_location</code>.</td>
<td><img src="image.png" alt="Diagram" /></td>
</tr>
<tr>
<td><code>busy: BOOLEAN</code> -- Is taxi busy?</td>
<td><img src="image.png" alt="Diagram" /></td>
</tr>
<tr>
<td><code>load (q: INTEGER)</code> -- Load <code>q</code> passengers.</td>
<td><img src="image.png" alt="Diagram" /></td>
</tr>
<tr>
<td><code>position: COORDINATE</code> -- Current position on map.</td>
<td><img src="image.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>
Inheriting features

deferred class

VEHICLE
inherit

MOVING
feature

[... Rest of class ...]
end

deferred 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.
**Inherited features**

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

- \( v \cdot \text{load}(...) \)
- \( e \cdot \text{take}(...) \)
- \( m \cdot \text{position} \quad \text{-- An expression} \)
- \( t \cdot \text{busy} \quad \text{-- An expression} \)
- \( e \cdot \text{load}(...) \)
- \( e \cdot \text{take}(...) \)
- \( e \cdot \text{position} \quad \text{-- An expression} \)
- \( e \cdot \text{busy} \quad \text{-- An expression} \)
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

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

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

More interesting:

\[\text{if some\_condition then}\]
\[v := \text{cab}\]
\[\text{else}\]
\[v := \text{tram}\]
\[\text{end}\]

A proper descendant type of the original
Assignments

Assignment:

\[ target := expression \]

So far (no polymorphism):

\[ expression \] was always of the same type as \( target \)

With polymorphism:

The type of \( expression \) is a descendant of the type of \( target \)
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.
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.
Conformance: full definition

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**
  - *

- **CLOSED FIGURE**
  - *

- **SEGMENT**
  - *

- **POLYLINE**
  - *

- **POLYGON**
  - *

- **ELLIPSE**
  - *

- **TRIANGLE**
  - *

- **RECTANGLE**
  - *

- **SQUARE**
  - *

- **CIRCLE**
  - *

* 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
      across vertex as v loop
        Result := Result + v[i].distance(v[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:

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

Permitted:

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

NOT permitted:

\[ x := p.\text{diagonal} \]  -- Even just after \( p := r \) !
\[ r := p \]
Dynamic binding

What is the effect of the following (if \textit{some\_test} is true)?

\begin{verbatim}
if some_test then
    p := r
else
    p := t
end
x := p.perimeter
\end{verbatim}

Redefinition: A class may change an inherited feature, as with \textit{POLYGON} redefining \textit{perimeter}.

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

Dynamic binding: Effect of \textit{p.perimeter} depends on run-time form of \textit{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 $x.f$)

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?

\[
\text{display (} f : \text{FIGURE}) \\
\hspace{1cm} \text{do} \hspace{1cm} \\
\hspace{2cm} \text{if} \hspace{0.2cm} \text{"} f \text{ is a CIRCLE" then} \hspace{1cm} \\
\hspace{3cm} \ldots \hspace{1cm} \\
\hspace{2cm} \text{elseif} \hspace{0.2cm} \text{"} f \text{ is a POLYGON" then} \hspace{1cm} \\
\hspace{3cm} \ldots \hspace{1cm} \\
\hspace{2cm} \text{end} \hspace{1cm} \\
\hspace{1cm} \text{end} \hspace{1cm}
\]

and similarly for all other routines!

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

With:

\[ f : \text{FIGURE} \]
\[ c : \text{CIRCLE} \]
\[ p : \text{POLYGON} \]

Initialize:

\[
\text{if } \ldots \text{ then } \\
\quad f := c \\
\text{else } \\
\quad f := p \\
\text{end}
\]

and:

\[
\text{create } c.\text{make}(\ldots) \\
\text{create } p.\text{make}(\ldots)
\]

Then just use:

\[
\begin{align*}
&f.\text{move}(\ldots) \\
&f.\text{rotate}(\ldots) \\
&f.\text{display}(\ldots)
\end{align*}
\]

-- and so on for every operation on \( f \)!
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 \textit{with} type safety

Dynamic binding: automatic adaptation of operation to target, for more modular software architectures
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
Redefinition 2: LINE_VEHICLE

defferred 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
Dynamic binding

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

\[ m: MOVING, l: LINE_VEHICLE, t: TAXI \]

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

\[ m.update\_coordinates \]

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.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
- Specialization
Extending the basic notion of class

Genericity

Inheritance
Conformance

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

fleet: LIST [VEHICLE]
v: VEHICLE

extend (v : G)

-- Add a new occurrence of v.

fleet.extend (v)
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

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

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

Constrained

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

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

\textbf{class} \texttt{LIST}[: G] \textbf{feature}

\textit{extend} (x : G) ... \\
\textit{last} : G ... \\
\textbf{end}

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

\textit{cities} : \texttt{LIST}[: \texttt{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 + 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

... Other features ...
Adding two vectors

\[ u + v = w \]
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)
    c := a + b -- Requires "+" operation on \( G \!
    Result.put (c, i)
    i := i + 1
end
\end{verbatim}
The solution

Declare class \textit{VECTOR} as

\[
\text{class } \textit{VECTOR} [G \rightarrow \textit{NUMERIC}] \text{ feature } \\
... \text{ The rest as before ... } \\
\text{end}
\]

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

Make _VECTOR_ itself a descendant of _NUMERIC_, effecting the corresponding features:

```plaintext
class VECTOR [G -> NUMERIC] inherit NUMERIC
  feature
    ... Rest as before, including _infix_ "+"...
  end
```

Then it is possible to define:

```plaintext
v: VECTOR [INTEGER]
vv: VECTOR [VECTOR [INTEGER]]
vvv: VECTOR [VECTOR [VECTOR [INTEGER]]]
```
Combining genericity with inheritance

Type parameterization

Abstraction

Genericity

Inheritance

Specialization

LIST_OF_CARS

SET_OF_CARS

LIST_OF_CITIES

LIST_OF_CARS

LINKED_LIST_OF_CARS

LIST_OF_PERSONS
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)
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:

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

defferred class COMMAND feature

done: BOOLEAN
   -- Has this command been executed?

execute
   -- Carry out one execution of this command.

ensure
   already: done

undo
   -- Cancel an earlier execution of this command.

require
   already: done

end
Command class hierarchy

**COMMAND**

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

**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

end

... also item, index, go_ith, put_left ...
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:

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

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

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

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

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

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

A polymorphic data structure:

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"
    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*

COMMAND

* deferred
+ effective

+ REMOVAL

execute* undo*
line: STRING
index: INTEGER

+ INSERTION

execute* undo*
index

...
Example hierarchy

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

- CLOSED FIGURE

- CENTER
- DISPLAY
- ROTATE
- PERIMETER
- DIAGONAL
- SIDE1
- SIDE2

* deferred
+ effective
++ redefined
Enforcing a type: the problem

```plaintext
fl.store(“FN”)
...

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

What’s wrong with this?

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

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

  `print (r.diagonal)`

  ... Do anything else with `r`, guaranteed

  ... to be non void and of dynamic type `RECTANGLE`

else

  `print ("Too bad.")`

end
Earlier mechanism: assignment attempt

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

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

\[ r ?= f \]

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

$x \neq y$

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. *LIST*:

```forth
forth
require not after
deferred
ensure
  index = old index
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
  from 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

- CONTAINER
  - BOX
  - CONTAINER
  - COLLECTION
    - BAG
    - SET
    - TRAVERSABLE
      - TRAVERSABLE
      - HIERARCHICAL
      - LINEAR
  - FINITE
  - INFINITE
  - BOUNDED
  - UNBOUNDED
  - COUNTABLE
  - RESIZABLE
  - INDEXABLE
  - TABLE
  - ACTIVE
    - INTEGER_INTERVAL
  - CURSOR_STRUCTURE
  - DISPENSER
  - SEQUENCE
  - STACK
  - QUEUE

* deferred
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

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

(total order relation)

```
+', '-', '...
```

(commutative ring)
How do we write *COMPARABLE*?

defered class *COMPARABLE* feature

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

\[\text{less\_equal\ alias } "\leq" (x : \text{COMPARABLE}) : \text{BOOLEAN}\]
do
\hspace{1cm} Result := (Current < x \text{ or } (Current = x))
end

greater\ alias "\textbf{>}" (x : \text{COMPARABLE}) : \text{BOOLEAN}
do Result := (x < \text{Current}) end

greater\_equal\ alias "\textbf{>=}" (x : \text{COMPARABLE}) : \text{BOOLEAN}
do Result := (x <= \text{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

...
Television station example

class SCHEDULE feature

    segments: LIST[SEGMENT]

dend

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
deferred
end

print
-- Produce paper version.
deferred
end
deferred
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.
invariant

in_list: (1 <= index) and (index <= schedule.segments.count)

in_schedule: schedule.segments.item(index) = Current

next_in_list: (next /= Void) implies

   (schedule.segments.item(index + 1) = next)

no_next_iff_last: (next = Void) = (index = schedule.segments.count)

non_negative_rating: rating >= 0

positive_times: (starting_time > 0) and (ending_time > 0)

sufficient_duration: ending_time - starting_time >= Minimum_duration

decent_interval:

   (next.starting_time) - ending_time <= Maximum_interval

end
Commercial

note
  description: "Advertizing segment"
deferred 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
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

   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 $C$:

if $a1.\alpha$ then

$a1.r(\ldots)$

-- 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
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
Warning

Forget all you have heard!

*Multiple inheritance is not* the works of the devil

*Multiple inheritance is not* bad for your teeth

(Even though Microsoft Word apparently does not like it:

Object-oriented programming would become a mockery of itself if it had to renounce multiple inheritance.)
This is **repeated**, not just multiple inheritance

Not the basic case!

(Although it does arise often; why?)
Another warning

The language part of this lecture are Eiffel-oriented

Java and C# mechanisms (single inheritance from classes, multiple inheritance from interfaces) will also be discussed

C++ also has multiple inheritance, but I will not try to describe it
Composite figures
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

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

- **COMPOSITE FIGURE**
  - **LIST [FIGURE]**
A composite figure as a list

Cursor

item

forth

after
Composite figures

class **COMPOSITE_FIGURE** inherit **FIGURE**

**LIST[FIGURE]**

feature

  **display**

  do
  
  across **Current** as **c** loop

  **c.item.display**

  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
  - STRING
  - REAL
  - INTEGER

- NUMERIC
  - COMPLEX
  - REAL
  - INTEGER

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

+, -, *, /,
(commutative ring)
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**: `<`, `<=`, `>`, `>=`, ...
  - (total order relation)
- **REAL**: `+`, `-`, `∗`, `/`
  - (commutative ring)
- **INTEGER**: `+`, `-`, `∗`, `/`
- **STRING**: `<`, `<=`, `>`, `>=`, ...
- **COMPLEX**
How do we write `COMPARABLE`?

defined class `COMPARABLE [G]` feature

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

```
less_equal alias "<=" (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 ">=" (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*
A common Eiffel library idiom

class ARRAYED_LIST [G] inherit
    LIST[G]
    ARRAY[G]

feature

    ... Implement LIST features using ARRAY features ... 
end

For example:

\[ i_{th} (i: INTEGER): G \]

\[ \text{-- Element of index 'i'.} \]

do
\[ \text{Result := item} (i) \]
end
Could use delegation instead

class ARRAYED_LIST [G] inherit LIST[G]

feature

  rep: LIST[G]

  ... Implement LIST features using ARRAY features applied to rep...

end

For example:

  \[ i_{\text{th}} (i: \text{INTEGER}): G \]

  -- Element of index \('i'\).

  do
    Result := rep \cdot \text{item} (i)
  end
Non-conforming inheritance

class

\textit{ARRAYED\_LIST} \{G\}

inherit

\textit{LIST} \{G\}

\textcolor{red}{\textbf{inherit}} \{\texttt{NONE}\}

\textit{ARRAY} \{G\}

feature

\ldots \text{ Implement } \textit{LIST} \text{ features using } \textit{ARRAY} \text{ features}

\ldots

end
Multiple inheritance: Name clashes
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:
- \( a1.A_f \)
- \( b1.A_f \)
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)
Another application of renaming

Provide locally better adapted terminology.
Example: *child (TREE); subwindow (WINDOW)*
Renaming to improve feature terminology

“Graphical” features: height, width, change_height, change_width, xpos, ypos, move...

“Hierarchical” features: superwindow, subwindows, change_subwindow, add_subwindow...

class WINDOW inherit

  RECTANGLE
  TREE [WINDOW]

  rename
    parent as superwindow,
    children as subwindows,
    add_child as add_subwindow

  end

feature ...
end

BUT: see style rules about uniformity of feature names
Feature merging

$A \xrightarrow{f^*} D \xrightarrow{f^*} B \xrightarrow{} C \xrightarrow{f^+} D$

* 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

* Deferred
+ Effective
\[\text{Renaming}\]
Feature merging: effective features

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

\[ f^- \quad f^- \quad f^- \quad f^- \]

* Deferred
+ Effective
-- Undefine
deferred class
  \( T \)
inherit
  \( S \)
  undefine v end

feature
  ...
end
Merging through undefinedness

class D
  inherit A
  undefine f end
end

feature ...
end

* Deferred
+ Effective
-- Undefine
Merging effective features with different names

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

\[ a1: A \quad b1: B \quad c1: C \quad d1: D \]

\[ a1.g \quad b1.f \quad c1.h \quad d1.f \]
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 a case of *repeated* inheritance.
Indirect and direct repeated inheritance
Multiple is also repeated inheritance

A typical case:

```
copy ++
is_equal ++

ANY

LIST

C

D

copy
is_equal

C_copy
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:

\[
a1 : \text{ANY} \\
d1 : \text{D} \\
\ldots \\
a1 := d1 \\
a.copy(...)\
\]
Removing the ambiguity

class \( D \)
inherit \( \text{LIST} \ [ \ T \] \)

select
\[
\begin{align*}
\text{copy}, \\
\text{is\_equal}
\end{align*}
\]
end

C
rename
\[
\begin{align*}
\text{copy as } C\_\text{copy}, \\
\text{is\_equal as } C\_\text{is\_equal},
\end{align*}
\]
...
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.
What we have seen

A number of games one can play with inheritance:

- Multiple inheritance
- Feature merging
- Repeated inheritance