Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Lecture 13: More Inheritance & Genericity
Our program for this lecture

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
(Reminder) A generic class

class LIST\([G]\) feature

\begin{align*}
\text{extend}(x : G) & \ldots \\
\text{last} : G & \ldots
\end{align*}

end

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

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

\[\text{cities : LIST[CITY]}\]
\[\text{people : LIST[PERSON]}\]
\[\text{c : CITY}\]
\[\text{p : PERSON}\]

... 

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)`
(Reminder) Using genericity

\textit{LIST [CITY]}
\textit{LIST [LIST [CITY]]}
...

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

(But every type remains \textit{based} on a class.)
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 *types*, the actual generic parameters, in brackets
  
  e.g. \( \text{LIST[METRO\_STATION]} \)
  
  e.g. \( \text{LIST[ARRAY[METRO\_STATION]]} \)
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

\[ u + v = w \]

\[ \begin{align*}
  a + b &= c \\
  2 + 1 &= 3
\end{align*} \]
Constrained genericity

Body of `plus alias "+"`:

```
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
```
The solution

Declare class `VECTOR` as

```plaintext
class VECTOR [G -> NUMERIC] feature
    ... The rest as before ...
end
```

Class `NUMERIC` (from the Kernel Library) provides features `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{align*}
v & : \texttt{VECTOR [INTEGER]} \\
v v & : \texttt{VECTOR [VECTOR [INTEGER]]} \\
v v v & : \texttt{VECTOR [VECTOR [VECTOR [INTEGER]]]} \end{align*}
Extending the basic notion of class

**Genericity**

**Inheritance**

![Diagram showing relationships between classes and data structures.](image)
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 ...
  end
  last: G
  ...
end

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

(POLYGON)  (CIRCLE)  (CIRCLE)  (ELLIPSE)  (POLYGON)
Another example hierarchy

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

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

...

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

Object-Test Local

if \{ r: RECTANGLE \} fl.retrieved("FILE_NAME") then

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

else

print("Too bad.")

end

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

\( f: \text{FIGURE} \)
\( r: \text{RECTANGLE} \)

... 

\( \text{fl.retrieve("FILE_NAME")} \)

\( f := \text{fl.last} \)

\( r := f \)

\begin{align*}
\text{if } r \neq \text{Void} \text{ then} & \\
& \text{println}(r.\text{diagonal}) \\
\text{else} & \\
& \text{println("Too bad.")}
\end{align*}

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. \texttt{LIST}:

\begin{verbatim}
forth
require not after
deferred
ensure \texttt{index = old index + 1}
end
\end{verbatim}
Mixing deferred and effective features

In the same class

\[
\text{search}(x: G)
\]

-- Move to first position after current
-- where \( x \) appears, or \( \text{after} \) if none.
do
from until \( \text{after} \) or else \( \text{item} = x \) 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 (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"

deferred class SCHEDULE feature

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

  air_time: DATE
    -- 24-hour period
      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: \((1 \leq index) \text{ and } (index \leq schedule.segments.count)\)

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

next_in_list: \((next /= Void) \text{ implies } (schedule.segments.item(index + 1) = next)\)

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

non_negative_rating: rating \geq 0

positive_times: \((starting_time > 0) \text{ and } (ending_time > 0)\)

sufficient_duration:

ending_time - starting_time \geq Minimum_duration

decent_interval:

\((next.starting_time) - ending_time \leq Maximum_interval\)

end
Commercial

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
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
  \textit{in\_valve}, \textit{out\_valve} : VALVE
  \begin{itemize}
    \item \texttt{in\_valve.open}
    \item \texttt{out\_valve.closed}
  \end{itemize}
  \texttt{deferred ensure}
  \begin{itemize}
    \item \texttt{in\_valve.closed}
    \item \texttt{out\_valve.closed}
    \item \texttt{is\_full}
  \end{itemize}
end

\textit{empty}, \textit{is\_full}, \textit{is\_empty}, \texttt{gauge}, \texttt{maximum}, ... [Other features] ...

invariant
  \texttt{is\_full} = (\texttt{gauge} \geq 0.97 \times \texttt{maximum}) \texttt{ and } (\texttt{gauge} \leq 1.03 \times \texttt{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 $C$: if $a1.\alpha$ then $a1.r(...)\quad\text{-- Here } a1.\beta \text{ holds}\end{if}

---

Correct call in $C$: if $a1.\alpha$ then $a1.r(...)\quad\text{-- Here } a1.\beta \text{ holds}\end{if}$
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

\[
\text{require else } \text{new_pre} \\
\text{ensure then } \text{new_post}
\]

Resulting assertions are:

- \( \text{original_precondition or new_pre} \)
- \( \text{original_postcondition and new_post} \)
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