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
   extend (x: G) ...
   last: G ...
end
```

To use the class: obtain a generic derivation, e.g.
```
cities: LIST[CITY]
```
(Reminder) Using generic derivations

```
cities : LIST[CITY]
people : LIST[PERSON]
c : CITY
p : PERSON
...
cities.extend (c)
people.extend (p)
```

```
c := cities.last
c.add_tram_line(Line8)
```

**STATIC TYPING:**
The compiler will reject:

- `people.extend (c)`
- `cities.extend (p)`

---

(Reminder) Using genericity

```
LIST [CITY]
LIST [LIST [CITY]]
```

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

(But every type remains 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. 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. \( \text{LIST[METRO\_STATION]} \)
  > \( \text{LIST[ARRAY[METRO\_STATION]]} \)

Genericity + inheritance 1: Constrained genericity

```lang-ocaml
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
  element ... Other features, in particular item to access an
  end
```

Adding two vectors

\[
\begin{align*}
U & + V = W \\
2 & \begin{array}{c}
\text{a} \\
\text{b} \\
\text{c}
\end{array} & + & \begin{array}{c}
\text{a} \\
\text{b} \\
\text{c}
\end{array} & = & \begin{array}{c}
\text{a} \\
\text{b} \\
\text{c}
\end{array}
\end{align*}
\]
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

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 VECTOR itself a descendant of NUMERIC, effecting the corresponding features:

class VECTOR [G => NUMERIC] inherit NUMERIC
  feature
    ... Rest as before, including infix "+" ...
  end

Then it is possible to define

v : VECTOR [INTEGER]
vv : VECTOR [VECTOR [INTEGER]]
vvv : VECTOR [VECTOR [VECTOR [INTEGER]]]
Extending the basic notion of class

Genericity + Inheritance 2: Polymorphic data structure

Example hierarchy
Forcing a type: the problem

```plaintext
fl.store("FILE_NAME")
...
-- Two years later:
fl := retrieved("FILE_NAME") - 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.
```

The solution: Assignment attempt

```plaintext
fl.store("FILE_NAME")
...
-- Two years later:
fl ?? retrieved("FILE_NAME")
x := fl.last -- [1]
print(x.diagonal) -- [2]

But:
If x is declared of type RECTANGLE, [1] is invalid.
If x is declared of type FIGURE, [2] is invalid.
```

Assignment attempt

```plaintext
f: FIGURE
r: RECTANGLE
...
fl.retrieve("FILE_NAME")
f := fl.last
r ?? f
if r /= Void then
  print(r.diagonal)
else
  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.

Preview: the Object Test

Expression to be tested

Available with EiffelStudio 6.1 (November 2007)

\[ \text{if } \{ r : \text{RECTANGLE} \} \text{ then}\]

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

else

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

end

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 + 1
end
```

Deferred!

In the same class -- Move to first position after current -- where x appears, or after if none.
```forth
search(x; i)
do from until after or else item = x loop forth end end
```

“Programs with holes”

Mixing deferred and effective features

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

“Don’t call us, we’ll call you!”
Intro. to Programming, lecture 14: More Inheritance & Genericity

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

Television station example

class SCHEDULE feature
  segments: LIST[SEGMENT]
end

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

Schedules

```plaintext
note
description: "24-hour TV schedule" deferred 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
```
Segment

```plaintext
note
description: "Details of a schedule"

defined 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

Segment (continued)

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_if_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

Commercial

note
description: "Advertising segment"
defined 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

invariant

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

set_primary(p: PROGRAM)
-- Attach commercial to p.
require
primary_exists: p /= Void
some_schedule: p.schedule = schedule
before: p.starting_time <= starting_time
defined as ensured:
index_updated: index + p.index
primary_updated: primary = p
```

Commercial (continued)
Chemical plant example

defered class VAT
inherirt TANK

feature
  in_valve, out_valve VALVE;
  full
    require
      in_valve open
      out_valve closed
    deferred
      in_valve closed
      out_valve closed
      is_full
    and
      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:
if \( a1.\alpha \) then
\( a1.r(...) \)
-- Here \( a1.\beta \) holds

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

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

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