Lecture 6: Object Creation
Contracts

Associated with an individual feature:
- **Preconditions**
- **Postconditions**

Associated with a class:
- **Class invariant**
Preconditions and postconditions

\(i\_th (i : \text{INTEGER}): \text{STATION}\)
-- The station of index \(i\) on this line
\textbf{require}
\hspace{1em} \text{not\_too\_small: } i \geq 1
\hspace{1em} \text{not\_too\_big: } i \leq \text{count}

\textbf{remove\_all\_stations}
-- Remove all stations except the South-West end.
\textbf{ensure}
\hspace{1em} \text{only\_one\_left: } \text{count} = 1
\hspace{1em} \text{both\_ends\_same: } \text{sw\_end} = \text{ne\_end}
Class invariants

The invariant expresses consistency requirements between queries of a class

\[
\text{invariant}
\begin{align*}
\text{southwest\_is\_first: } & \quad \text{sw\_end} = \text{i\_th (1)} \\
\text{northeast\_is\_last: } & \quad \text{ne\_end} = \text{i\_th (count)}
\end{align*}
\]
Invariant principle

A class invariant must hold as soon as an object is created, then before and after the execution of any of the features of the class available to clients.

- Preconditions must hold at beginning of feature
- Postconditions must hold at end of feature
- Class invariants must hold at both points
Contracts

A contract is a semantic condition characterizing usage properties of a class or a feature
Design by Contract

Analysis, design, implementation:  
*Get the software right from the start*

Testing, debugging, quality assurance

Management, maintenance/evolution

Inheritance

Documentation
Creating objects

In previous examples, *Paris, Line8* etc. denote predefined objects. We’ll now create objects ourselves.

Fictitious metro line, *fancy_line*:
Example: **LINE_BUILDING**

```plaintext
class LINE_BUILDING inherit TOUR
  feature
    build_a_line is
      -- Build an imaginary line and highlight it on the map.
      do
        Paris.display
        Metro.highlight
        -- "Create fancy_line and fill in its stations"
        fancy_line.illuminate
      end
  end
end
fancy_line: METRO_LINE
  -- An imaginary line of the Metro
```
Identifiers, entities, variables

An **identifier** is a name chosen by the programmer to represent certain program elements.
It may denote:

- A class, e.g. **METRO_STATION**
- A feature, e.g. **i_th**
- A run time value, such as an object or object reference, e.g. **fancy_line**

An identifier that denotes a run-time value is called an **entity**, or a **variable** if it can change its value.

During execution an entity may become **attached** to an object.
Entity attached to an object

In the program: an entity
In memory, during execution: an object

ENTITY

fancy_line

(LINE_BUILDING)

OBJECT

reference

FIELDS

(METRO_LINE)

Generating class

Generating class
class LINE_BUILDING inherit TOUR

feature

    build_a_line is
    -- Build an imaginary line and highlight it on the map.
    do
        Paris.display
        Metro.highlight
        -- "Create fancy_line and fill in its stations"
        fancy_line.illuminate
    end

    fancy_line: METRO_LINE
    -- An imaginary line of the Paris Metro
end
Initial state of a reference

In an instance of `LINE_BUILDING`, may we assume that `fancy_line` is attached to an instance of `METRO_LINE`?

This object has been created

Where does this one come from?

```
fancy_line

( LINE_BUILDING )

reference

( METRO_LINE )
```
By default

Initially, *fancy_line* is not attached to any object: its value is a **void** reference.
States of a reference

During execution, a reference is either:

- Attached to a certain object
- Void

- To denote a void reference: use \texttt{Void}
- To find out if \( x \) is void, use the condition
  \[ x = \texttt{Void} \]
- Inverse condition (\( x \) is attached to an object):
  \[ x \neq \texttt{Void} \]
The trouble with void references

The basic mechanism of computation is feature call

Apply feature $f$

$x.f(a, \ldots)$

To object to which $x$ is attached

Possibly with arguments

Since references may be void, $x$ might be attached to no object

The call is erroneous in such cases
Example: call on void target

class LINE_BUILDING inherit TOUR

feature
  build_a_line is
  -- Build an imaginary line and highlight it on the map.
  do
    Paris.display
    Metro.highlight
    -- “Create fancy_line and fill in its stations”
    fancy_line.illuminate
  end

fancy_line : METRO_LINE
  -- An imaginary line of the Paris Metro
end
Exceptions

Abnormal event during execution. Examples:

- Call on void reference: `fancy_line.illuminate`
- Attempt to compute $a / b$ where $b$ has value 0.

A failure will happen unless the program has code to recover from the exception ("rescue" clause in Eiffel, "catch" in Java)

Every exception has a type, appearing in EiffelStudio run-time error messages, e.g.

- Feature call on void reference
- Arithmetic underflow
Creating objects explicitly

To avoid exception:

Change the procedure *build_a_line* to create an object and attach it to *fancy_line* before call to *illuminate*.
Why do we need to create objects?

Shouldn’t we assume that a declaration

\[
fancy\_line: METRO\_LINE
\]

creates an instance of \textit{METRO\_LINE} and attaches it to \textit{fancy\_line}?

(Answer in a little while...)
class LINE_BUILDING inherit TOUR

feature

  build_a_line is
  -- Build an imaginary line and highlight it on the map.
  do
    Paris.display
    Metro.highlight
    -- "Create fancy_line and fill in its stations"
    fancy_line.illuminate
  end

  fancy_line: METRO_LINE
  -- An imaginary line of the Paris Metro
end
Creating simple objects

To create *fancy_line*, we need to create objects representing stations and stops of the line.

Need instances of not only *METRO_STATION* but *STOP* (why?)

What is the next stop after Concorde?

Madeleine?

Musee du Louvre?
An instance of `STOP` has:

- Reference to a station; must be non-void.
- Reference to next stop; void at the end

```
( STOP )
```

```
( METRO_STATION )
```
Interface of class **SIMPLE_STOP**

```plaintext
class SIMPLE_STOP feature

station: METRO_STATION
    -- Station which this stop represents

next: SIMPLE_STOP
    -- Next stop on the same line

set_station (s: METRO_STATION)
    -- Associate this stop with s.
    require
        station_exists: s /= Void
    ensure
        station_set: station = s

link (s: SIMPLE_STOP)
    -- Make s the next stop on the line.
    ensure
        next_set: next = s

end
```
class LINE_BUILDING inherit TOUR

feature

    build_a_line

    -- Build an imaginary line and highlight it on the map.
    do
    Paris.display
    Metro.highlight
    -- "Create fancy_line and fill in its stations"
    fancy_line.illuminate
    end

fancy_line: METRO_LINE

    -- An imaginary line of the Paris Metro

end
Creating an instance of **SIMPLE_STOP**

class **LINE_BUILDING** inherit
   TOUR

feature

**stop1**: SIMPLE_STOP
   -- First stop on the line

**build_a_line** is
   -- Build an imaginary line and highlight it on the map.
   do
   Paris.display
   Metro.highlight
   create **stop1**
   -- "Create more stops and finish building fancy_line"
   fancy_line.illuminate
   end

**fancy_line**: METRO_LINE
   -- An imaginary line of the Paris Metro

end
Creation instruction

Basic operation to produce objects at run time:

- Create new object in memory
- Attach entity to it

![Diagram showing object creation with actions: create stop1, next, station, SIMPLE_STOP]
Type of created objects

Every entity is declared with a certain type:

\[ \text{stop1: SIMPLE\_STOP} \]

A creation instruction

\[ \text{create stop1} \]

produces, at run time, an object of that type.
We declare three stops:

\(\text{stop1, stop2, stop3 : SIMPLE\_STOP}\)
build_a_line

build_a_line is
    -- Build an imaginary line and highlight it on the map.
    do
        Paris.display
        Metro.highlight
        -- Create the stops and associate each to its station:
        create stop1
            stop1.set_station(Station_Montrouge)
        create stop2
            stop2.set_station(Station_Issy)
        create stop3
            stop3.set_station(Station_Balard)
        -- Link each applicable stop to the next:
            stop1.link(stop2)
            stop2.link(stop3)
        -- “Create fancy_line and give it the stops just created”
            fancy_line.illuminate
    end
Why do we need to create objects?

Shouldn’t we assume that a declaration

\[ \text{fancy\_line} \colon \text{METRO\_LINE} \]

creates an instance of \text{METRO\_LINE} and attaches it to \text{fancy\_line}?
Void references are useful

Married persons:

(Person)

(Person)

spouse

spouse
Void references are useful

Unmarried person:
Void references are useful

Even with married persons...

... we shouldn’t create an object for **spouse** every time we create an instance of **PERSON** (why?)
Using void references

Create every *PERSON* object with a void *spouse*
Using void references

Create every \textit{PERSON} object with a void \textit{spouse}

\begin{center}
\includegraphics[width=\textwidth]{diagram.png}
\end{center}
Using void references

Create every `PERSON` object with a void `spouse` ...

... then attach the `spouse` references as desired, through appropriate instructions
Using void references

Create every \textit{PERSON} object with a void \textit{spouse} ...

\begin{figure}
\centering
\includegraphics[width=0.7\textwidth]{diagram.png}
\caption{Diagram of object creation and spouse references.}
\end{figure}

... then attach the \textit{spouse} references as desired, through appropriate instructions
References to linked structures

Last \textit{next} reference is void to terminate the list.
 build_a_line

build_a_line is
    -- Build an imaginary line and highlight it on the map.
    do
        Paris.display
        Metro.highlight
        -- Create the stops and associate each to its station:
        create stop1
        stop1.set_station(Station_Montrouge)
        create stop2
        stop2.set_station(Station_Issy)
        create stop3
        stop3.set_station(Station_Balard)
        -- Link each applicable stop to the next:
        stop1.link(stop2)
        stop2.link(stop3)
        -- "Create fancy_line and give it the stops just created"
        fancy_line.illuminate
    end
The need for creation procedures

Creating and initializing a \textit{SIMPLE\_STOP} object:

\begin{verbatim}
create some_stop
some_stop.set_station(existing_station)
\end{verbatim}

Invariant of the class:

\begin{verbatim}
invariant
station_exists: station /= Void
\end{verbatim}

After creation: invariant not satisfied!
Creation procedures

Declare `set_station` as a creation procedure and merge initialization with creation:

```
create new_stop1.set_station(Station_montrouge)
-- Same effect as previous two instructions
```

- **Convenience**: initialize upon creation
- **Correctness**: ensure invariant right from the start
**STOP**

```plaintext
class STOP
    create
        set_station
    feature
        station: METRO_STATION
            -- Station which this stop represents
        next: STOP
            -- Next stop on the same line
        set_station (s: METRO_STATION)
            -- Associate this stop with s.
            require
                station_exists: s /= Void
            ensure
                station_set: station = s
        link (s: STOP)
            -- Make s the next stop on the line.
            ensure
                next_set: next = s
    invariant
        station_exists: station /= Void
end
```

List zero or more creation procedures

Now doubles up as creation procedure
Creation principle

If a class has a non-trivial invariant, it must list one or more creation procedures, whose purpose is to ensure that every instance, upon execution of a creation instruction, will satisfy the invariant.

This allows the author of the class to force proper initialization of all instances that clients will create.
Creation procedures

Useful even in the absence of a strong invariant to combine creation with initialization:

```plaintext
class POINT create	default_create, make_cartesian, make_polar
feature
...
end
```

Inherited by all classes, by default does nothing

Valid creation instructions:
create your_point.default_create
create your_point
create your_point.make_cartesian (x, y)
create your_point.make_polar (r, t)
Object creation: summary

To create an object:

- If class has no `create` clause, use basic form, `create x`

- If the class has a `create` clause listing one or more procedures, use `create x.make (...)`

where `make` is one of the creation procedures, and “(...)“ stands for arguments if any
Correctness of an instruction

For every instruction we must know precisely, in line with the principles of Design by Contract:

- How to use the instruction correctly: its precondition.
- What we are getting in return: the postcondition.

Together, these properties (plus the invariant) define the correctness of a language mechanism.

What is the correctness rule for a creation instruction?
Correctness of a creation instruction

**Creation Instruction Correctness Rule**

**Before** creation instruction:

1. Precondition of its creation procedure, if any, must hold

**After** creation instruction with target \( x \) of type \( C \):

2. \( x \neq \text{Void} \) holds
3. Postcondition of creation procedure holds
4. Object attached to \( x \) satisfies invariant of \( C \)
Successive creation instructions

The correctness condition does not require $x$ to be void:

```
create x
  -- Here $x$ is not void
create x
```

![Diagram showing the creation of an object](image)
Effect of creation instruction

- won’t be void after creation instruction (whether or not it was void before)

If there is a creation procedure, its postcondition will hold for newly created object

The object will satisfy the class invariant
How it all starts

Executing a system consists of creating a root object, which is an instance of a designated class from the system, called its root class, using a designated creation procedure of that class, called its root procedure.

Root creation procedure may:

- Create new objects
- Call features on them, which may create other objects
- Etc.
Executing a system

Root object

Root procedure

create obj1.r1
create obj2.r2

Intro. to Programming, lecture 6: Object creation
Current object

At any time during execution, there is a current object, on which the current feature is being executed.

Initially it is the root object.

During a “qualified” call $x.f(a)$, the new current object is the one attached to $x$.

At the end of such a call, the previous current object resumes its role.
The design process

A system is a particular assembly of certain classes, using one of them as root class.

The classes may have value of their own, independently of the system: they may be reusable.
Extendibility & reusability

- **Extendibility**: the ease with which it is possible to adapt the system to changing user needs

- **Reusability**: the ease of reusing existing software for new applications

Older approaches to software engineering, based on the notion of main program and subprograms, pay less attention to these needs.
Specifying the root

How to specify the root class and root creation procedure of a system?

Use EiffelStudio
What we have seen

- Class invariants
- Concept of Design by Contract
- The notion of exception
- Object creation
- Creation procedures
- Relationship between creation procedures and invariants
- System execution
End of lecture 6