Lecture 4: The Interface of a Class
Client, supplier

Definitions

• A **client** of a software mechanism is a system of any kind — such as a software element, a non-software system, or a human user — that uses it.

• For its clients, the mechanism is a **supplier**.
Picturing the client relation

(See diagram tool of EiffelStudio.)
An interface of a set of software mechanisms is the description of techniques enabling clients to use these mechanisms.
Kinds of interface

**User interface: when the clients are people**

- **GUI**: Graphical User Interface
- **Text interfaces, command line interfaces.**

**Program interface: the clients are other software**

- **API**: Application Programming Interface
  (or: Abstract Programming Interface)

We’ll now study class APIs.
A user interface (GUI)
Classes

An object (previous lecture) is a software machine allowing programs to access and modify a collection of data

Examples:
- A city
- A tram line
- An element of the GUI such as a button

Each object belongs to a certain class, defining the applicable operations, or features

Example:
- The class of all cities
Definitions: class, instance, generating class

**Class**

A class is the description of a set of possible run-time objects to which the same features are applicable.

A class represents a category of things. An object represents one of these things.

**Instance, generating class**

If an object \( O \) is one of the objects described by a class \( C \), then \( O \) is an instance of \( C \), and \( C \) is the generating class of \( O \).
Objects vs. classes

Classes exist only in the **software text:**
- Defined by class text
- Describes properties of associated instances

Objects exist only during **execution:**
- Visible in program text through names **denoting** run-time objects
  - Example: *Paris*
Finding appropriate classes is a central part of software design

(the organization of the architecture of a program)

Writing down the details is part of implementation
A class interface

In this discussion “interface” means API (not user interface).

We now look at interface of SIMPLE_LINE (simplified version of METRO_LINE)

This will be shown through EiffelStudio (use “Interface” button)
A query: “count”

How long is this line? See query \textit{count}

\texttt{count: INTEGER}

\-- Number of stations on this line

\textbf{Header comment:} states purpose of feature

\textit{“this line”}: the instance of \texttt{SIMPLE\_LINE} to which \textit{count} is applied

\textbf{Form of a query declaration:}

\begin{verbatim}
 feature_name: RETURN_TYPE
\end{verbatim}

\texttt{INTEGER}: a type denoting integer values (e.g. -23, 0, 256)
Style rule: header comments

Don't even *think* of writing a feature without immediately including a header comment explaining what it's about.
Expressions and their types

At run time, every object has a type: its generating class. Examples:

- **SIMPLE_LINE** for the object denoted by `Line8`
- **INTEGER** for the object denoted by `Line8.count`

In the program text, every expression has a type. Examples:

- **SIMPLE_LINE** for `Line8`
- **INTEGER** for `Line8.count`
Another query: \(i_{th}\)

What is the \(i\)-th station of the line? Feature \(i_{th}\).

Convention for consistency: numbering starts at south end

\(i_{th}(i: \text{INTEGER}): \text{METRO\_STATION}\)

-- The station of index \(i\) on this line
Two more queries

Which are the station at the ends of the line?

- `south_end: METRO_STATION`
  -- End station on south side
- `north_end: METRO_STATION`
  -- End station on north side

Properties of every line `l`:

- `l.south_end = l.i_th(1)`
- `l.north_end = l.i_th(l.count)`
Example: class QUERIES

class QUERIES inherit TOURISM

feature

    explore_on_click

        -- Try queries on lines.

        do

            Paris.display

            Console.show (Line8.count)

            Console.show (Line8.i_th (1))

            Console.show (Line8.i_th (Line8.count))

        end

end
A command: *remove_all_stations*

We want to rebuild *Line8* from scratch. We start by removing all stations:

```
remove_all_stations
    -- Remove all stations except south end.
```

Notes:

- Our metro lines always have at least one station, even after *remove_all_stations*
- If there is only one station, it is the value of both *south_end* and *north_end*
Command *extend*

Adding stations to a line:

```
extend(s: METRO_STATION)
-- Add s at end of this line.
```
Class **COMMANDS**

class **COMMAND** inherit
  **TOURISM**
feature
  **explore_on_click**
    -- Recreate a partial version of Line 8.
    do
      Line8.remove_all_sections
      -- No need to add Station_Balard, since
      -- remove_all_sections retains the south end.
      Line8.extend(Station_Lourmel)
      Line8.extend(Station_Boucicaut)
      Line8.extend(Station_Felix_Faure)
      -- We stop adding stations, to display some results:
      Console.show(Line8.count)
      Console.show(Line8.north_end.name)
    end
end
Defining proper interfaces

Not every feature is applicable to every possible argument and instance

Example: Line8.\textit{i\_th}(200) is wrong!

The class interface must be precise enough to convey such usage information
First try...

Add information to the header comment:

```plaintext
i_th (i: INTEGER): METRO_STATION

-- The i-th station on this line

-- (Warning: use only with i between 1 and count, inclusive.)
```

Better, but still not good enough:

- A comment is just an informal explanation
- The constraint needs a more official status in the interface
Contracts

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

Three principal kinds:

- Precondition
- Postcondition
- Class invariant
**Precondition**

Property that a feature imposes on every client:

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

-- The \( i \)-th station on this line

**require**

- not\_too\_small: \( i \geq 1 \)
- not\_too\_big: \( i \leq \text{count} \)

A feature with no **require** clause is always applicable, as if it had

**require**

- always\_OK: True
not_too_small: \( i \geq 1 \)
Precondition principle

A client calling a feature must make sure that the precondition holds before the call.

A client that calls a feature without satisfying its precondition is faulty (buggy) software.
Contracts

Contracts for debugging

Contracts for interface documentation
Postconditions

Precondition: obligation for clients
Postcondition: benefit for clients

\textbf{remove\_all\_stations}

\begin{verbatim}
  -- Remove all stations except the South-West end.
  ensure
  only_one_left: count = 1
  both_ends_same: south_end = north_end
\end{verbatim}

\textbf{extend}(s: METRO\_STATION)

\begin{verbatim}
  -- Add s at end of line.
  ensure
  new_station_added: \texttt{i\_th(count)} = s
  added_at_north: north_end = s
  one_more: count = old\_count + 1
\end{verbatim}
old notation

Usable in postconditions only

Denotes value of an expression as it was on routine entry

Example (in a class ACCOUNT)

\[\text{balance}: \text{INTEGER}\]
\[\quad \text{-- Current balance.}\]

\[\text{deposit (v: INTEGER)}\]
\[\quad \text{-- Add v to account.}\]
\textbf{require}
\[\quad \text{positive: } v > 0\]
\textbf{do}
\[\quad \ldots\]
\textbf{ensure}
\[\quad \text{added: } balance = \text{old balance} + v\]
\textbf{end}
A feature must make sure that, if its precondition held at the beginning of its execution, its **postcondition** will hold at the end.

A feature that fails to ensure its postcondition is buggy software.
What we have seen

- Classes
- Objects
- The notion of interface
- GUI vs API
- Commands & Queries
- Contracts: preconditions & postconditions
- Using contracts for debugging