Lecture 4: The Interface of a Class
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 Program Interface
  (or: Abstract Program Interface)

We’ll now study class APIs.
Classes

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

Examples objects may represent:

- A city
- A tram line
- A route through the city
- 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
- The class of all buttons
- etc.
Definition: 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
Definitions: class, instance, generating class

**Instance, generating class**

If an object \( O \) is one of the objects described by a class \( C \):

- \( O \) is an **instance** of \( C \)
- \( C \) is the **generating class** of \( O \)

A **class** represents a category of things

An **object** represents one of these things
Objects vs. classes

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

Objects exist only during **execution**:  
- Visible in program text through names **denoting** run-time objects

Example: *Paris*
Software construction

Finding appropriate classes is a central part of software design

(the development 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 LINE)

This will be shown through EiffelStudio (use “Interface” button)

Remember to add the TRAFFIC_ prefix to all the classes from the Traffic library. Use TRAFFIC_LINE and TRAFFIC_SIMPLE_LINE in EiffelStudio.
A query: “count”

How long is this line? See query count

`count : INTEGER`

-- Number of stations on this line

Header comment: states purpose of feature

“this line”: the instance of `SIMPLE_LINE` to which `count` is applied

Form of a query declaration:

```
feature_name : RETURN_TYPE
```

`INTEGER`: a type denoting integer values (e.g. -23, 0, 256)
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:

- **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:

- **LINE** for `Line8`
- **INTEGER** for `Line8.count`
Another query: \textit{i\_th}

What is the \textit{i}-th station of the line? Feature \textit{i\_th}.

Convention for consistency: numbering starts at south end.

\textit{i\_th}(i: \text{INTEGER}): \text{STATION}

-- The station of index \textit{i} on this line
Two more queries

Which are the station at the ends of the line?

\[
\begin{align*}
\text{south\_end: } & \text{STATION} \\
& \text{-- End station on south side} \\
\text{north\_end: } & \text{STATION} \\
& \text{-- End station on north side}
\end{align*}
\]

Properties of every line \( l \):

- \( l.\text{south\_end} = l.\text{i\_th}(1) \)
- \( l.\text{north\_end} = l.\text{i\_th}(l.\text{count}) \)
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_segments
-- Remove all stations except south end.
```

Notes:

- Our metro lines always have at least one station, even after *remove_all_segments*
- 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: 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_segments
                -- No need to add Station_Balard, since
                -- remove_all_segments 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: \texttt{Line8.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:

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

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

-- (Warning: use only with \( i \) between 1 and \text{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
Property that a feature imposes on every client:

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

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

\[
\text{require}
\]

not_too_small: \( i \geq 1 \)
not_too_big: \( i \leq \text{count} \)

The precondition of \( i_{th} \)

A feature with no \textbf{require} clause is always applicable, as if it had

\[
\text{require}
\]

always_OK: \textbf{True}
not_too_small: \( i \geq 1 \)
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

remove_all_segments

-- Remove all stations except the South-West end.

ensure

only_one_left: count = 1
both_ends_same: south_end = north_end

extend(s: STATION)

-- Add s at end of line.

ensure

new_station_added: \text{\texttt{i\_th(count)} = s}
added_at_north: north_end = s
one_more: count = old_count + 1

Expression value captured on entry
**old notation**

Usable in postconditions only

Denotes value of an expression as it was on routine entry

Example (in a class `ACCOUNT`):

```plaintext
balance: INTEGER
    -- Current balance.

deposit (v: INTEGER)
    -- Add v to account.
    require
        positive: v > 0
    do
        ...
    ensure
        added: \( \text{balance} = \text{old balance} + v \)
end
```
Postcondition principle

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
Reading assignment for next week

Chapters 1 to 6

Particularly: read chapter 5 (Logic), as I will go quickly through the parts also covered in “Discrete mathematics” (Prof. Ueli Maurer) and focus on the applications specific to programming