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.)

CLIENT SUPPLIER

Interface

Definition

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.
Intro. to Programming, lecture 4: the interfaces of a class

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

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.

Definition: 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 runtime objects, e.g. Paris

Software construction

Finding appropriate classes is a central part to 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 TRAFFIC_LINE

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

How long is this line? See query count

```
count: INTEGER
   -- Number of stations in this line
```

Header comment states purpose of feature

“This line”: the instance of LINE to which count is applied

Query declaration:

- Form: feature_name: RETURN_TYPE
  
  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:

- TRAFFIC_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:

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

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

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

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

Convention for consistency: Numbering starts at Southwest end

Two more queries

Which are the station at the ends of the line?

\[
\begin{align*}
sw\_end &: STATION \\
ne\_end &: STATION
\end{align*}
\]

-- End station on South or West side

-- End station on North or East side

Properties of every line \(l\):

\[
\begin{align*}
l\text{.sw}\_end &= l\_i_{th} (1) \\
l\text{.ne}\_end &= l\_i_{th} (l\text{.count})
\end{align*}
\]

Example: class QUERIES

class QUERIES inherit TOURISM

feature explore_on_click is

-- Test 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_segments

We want to rebuild Line8. We start by removing all stations: Command remove_all_stations

remove_all_stations

-- Remove all stations except South-West 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 sw_end and ne_end

Command extend_place

Adding stations to a line:

extend_place(s: STATION)

-- Add s at end of this line.

Class COMMANDS

class COMMAND inherit TOURISM

feature explore_on_click is
    -- Recreate a partial version of Line8.
    do
        Paris.display
        Line8.highlight
        Line8.remove_all_sections
        -- No need to add Station_Balard, since remove_all_sections retains the SW end.
        Line8.extend_place
        (Place_la_motte_picquet_grenelle)
        Line8.extend_place(Place_invalides)
        Line8.highlight
    end

end
Defining proper interfaces

Not every feature is applicable to every possible argument and instance

Example: 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:

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

\( \quad \text{-- The station of index } i \text{ on this line} \)

\( \quad \text{-- (Warning: use only with } i \text{ between 1 and count, inclusive.)} \)

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:

```plaintext
l_th (i: INTEGER; STATION
= The station of index i on this line
require
not too small: i >= 1
not too big: i <= count

The precondition of l_th
```

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

```plaintext
require
always_OK: True
```

assertions

```plaintext
assertion tag

not too small: i >= 1

Assertion
```

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

```
remove_all_stations
-- Remove all stations except the South-West end.
ensure
  only_one_left: count = 1
  both_ends_same: sw_end = ne_end

extend_place (s: STATION)
-- Add s at end of line.
ensure
  new_station_added: i_th (count) = s
  added_at_ne: ne_end = s
  one_more: count = old count + 1
```

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

End of lecture 4