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.

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)

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

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, e.g. Paris

Classes

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

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 $O$.

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).

Another query: \textit{i\_th}

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

```
\_th (i: INTEGER): STATION
-- The station of index \textit{i} on this line
```

Convention for consistency: Numbering starts at Southwest end

Style rule: header comments

Don’t even \textit{think} of writing a feature without immediately including a header comment explaining what it’s about.

Two more queries

Which are the station at the ends of the line?

```
sw_end: STATION
-- End station on South or West side
ne_end: STATION
-- End station on North or East side
```

Properties of every line \texttt{l}:
- \texttt{l.sw\_end} = \texttt{l\_th (1)}
- \texttt{l.ne\_end} = \texttt{l\_th (l.count)}

Expressions and their types

At run time, every object has a type: its generating class. Examples:
- \texttt{TRAFFIC\_LINE} for the object denoted by \texttt{Line8}
- \texttt{INTEGER} for the object denoted by \texttt{Line8.count}

In the program text, every expression has a type. Examples:
- \texttt{TRAFFIC\_LINE} for \texttt{Line8}
- \texttt{INTEGER} for \texttt{Line8.count}

Example: class \texttt{QUERIES}

```java
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
```

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

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

\texttt{remove\_all\_stations}

Notes:
- Our metro lines always have at least one station, even after
  \texttt{remove\_all\_stations}
- If there is only one station, it is the value of both \texttt{sw\_end} and
  \texttt{ne\_end}

Command extend_place

Adding stations to a line:

\texttt{extend\_place}(s; \texttt{STATION})

-- Add \texttt{s} at end of this line.

Class COMMANDS

\begin{verbatim}
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 \texttt{Station_Balard}, since
      -- \texttt{remove\_all\_sections} retains the SW end.
      Line8.extend_place(Place_la_motte_picquet_grenelle)
      Line8.extend_place(Place_invalides)
      Line8.highlight
    end
\end{verbatim}

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:

\texttt{i\_th(i; \texttt{INTEGER}; \texttt{STATION})}

-- The station of index \texttt{i} on this line

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:

\[
\textit{i\_th}\ (i: \text{INTEGER}; \text{STATION})
\rightarrow \text{The station of index } i \text{ on this line}
\]

\[\text{require}
\begin{align*}
\text{not\_too\_small}: & i \geq 1 \\
\text{not\_too\_big}: & i \leq \text{count}
\end{align*}
\]

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

\[\text{require}
\begin{align*}
\text{always\_OK}: & \text{True}
\end{align*}
\]

**Contrasts**

Contracts for debugging

Contracts for interface documentation

**Assertions**

Assertion tag

\[
\textit{not\_too\_small}: i \geq 1
\]

Assertion

**Postconditions**

Precondition: obligation for clients

Postcondition: benefit for clients

\[
\text{remove\_all\_stations}
\rightarrow \text{Remove all stations except the South-West end.}
\]

\[
\text{ensure}
\begin{align*}
\text{only\_one\_left}: & \text{count} = 1 \\
\text{both\_ends\_same}: & \text{sw\_end} = \text{ne\_end}
\end{align*}
\]

\[
\text{extend\_place\ (s: \text{STATION})}
\rightarrow \text{Add } s \text{ at end of line.}
\]

\[
\text{ensure}
\begin{align*}
\text{new\_station\_added}: & i\_th\ (\text{count}) = s \\
\text{added\_at\_ne}: & \text{ne\_end} = s \\
\text{one\_more}: & \text{count} = \text{old\ count} + 1
\end{align*}
\]

**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.

**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