Introduction to Programming

Lesson 3: The interface of a class

(With material from the ETH Zurich course “Introduction to Programming”)

Victor Rivera

August 30, 2016

News (1)

News (1)

Grading scheme (it can be found in Moodle):

News (1)

Grading scheme (it can be found in Moodle):

- Midterm 40% (Week 8 – October 4th, 2016)

News (1)

Grading scheme (it can be found in Moodle):

- Midterm 40% (Week 8 – October 4th, 2016)
- Final Exam 40%

News (1)

Grading scheme (it can be found in Moodle):

- Midterm 40% (Week 8 – October 4th, 2016)
- Final Exam 40%
- Assignments 20%

News (1)

Grading scheme (it can be found in Moodle):

- Midterm 40% (Week 8 – October 4th, 2016)
- Final Exam 40%
- Assignments 20%
    - Assignment-1 6%,
    - Assignment-2 6% and
    - Assignment-3 8%

News (1)

Grading scheme (it can be found in Moodle):

- Midterm 40% (Week 8 – October 4th, 2016)
- Final Exam 40%
- Assignments 20%
  - Assignment-1 6%,
  - Assignment-2 6% and
  - Assignment-3 8%

Quizzes (almost) bi-weekly (they are not going to be graded).

News (2)

News (2)

- Assignment 1 (Moodle)

News (2)

- Assignment 1 (Moodle)
- Quiz 1 (Moodle)
  - it will be available after this Lecture until tomorrow at 18h.
  - it takes around 10 mins.

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

Interface: Definition

An *interface* of a set of software mechanisms is the description of techniques enabling clients to use these mechanisms.

Kinds of interface

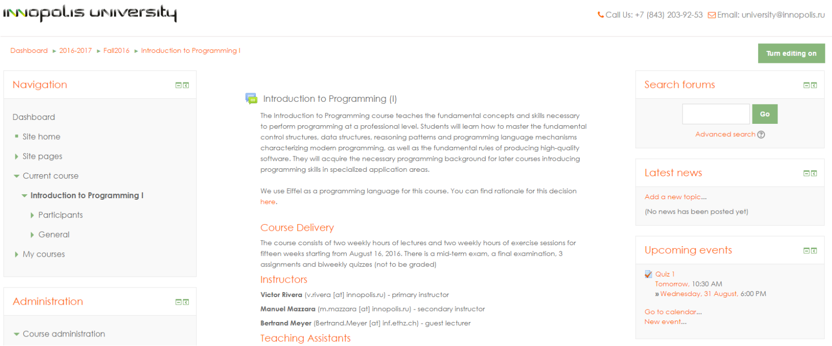
User interface: when the clients are people

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

Program interface: the clients are other software

- API: Application Program Interface (or: Abstract Program Interface)

A user interface (GUI)



Classes

An *object* 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.

Class: 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

More definitions ...

Instance and 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**

More definitions . . .

Instance and 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

Software construction

Finding appropriate classes is a central part of *Object Oriented Software Design*

Software construction

Finding appropriate classes is a central part of *Object Oriented Software Design* (the development of the architecture of a program).

Software construction

Finding appropriate classes is a central part of *Object Oriented Software Design* (the development of the architecture of a program). Writing down the details is part of *implementation*.

Design by Contract

Contracts

A *contract* is a semantic condition characterising usage properties of a class or a feature.

Three principal kinds:

- Precondition
- Postcondition
- Class invariant

Precondition

Property that *a feature imposes on every client* in order to function properly:

Precondition

Property that *a feature imposes on every client* in order to function properly:

```
i-th (i: INTEGER)
        -- The 'i'-th station on this line
    require
        not_too_small: i >= 1
        not_too_big: i <= count
...
```

Precondition

Property that *a feature imposes on every client* in order to function properly:

```
i-th (i: INTEGER)
        -- The 'i'-th station on this line
    require
        not_too_small: i >= 1
        not_too_big: i <= count
...
```

The precondition of i_th

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

```
require
    always_tok: true
```

not_too_small: $i >= 1$

Assertions

$$\underbrace{\text{not\_too\_small}: \qquad i >= 1}_{\text{Assertion}}$$

Assertions

$$\overbrace{\text{not\_too\_small}}^{\text{Assertion Tag}} \underbrace{}_{} $$

$$\underbrace{\overbrace{\text{not\_too\_small}}^{\text{Assertion Tag}} : \quad i >= 1}_{\text{Assertion}}$$

Assertions

$$\underbrace{\overbrace{\text{not\_too\_small}}^{\text{Assertion Tag}} : \overbrace{\quad i >= 1 \quad}^{\text{Condition}}}_{\text{Assertion}}$$

Precondition principle

Precondition principle

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

Precondition principle

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.

What are preconditions in practice?

- *Boolean expressions* checked every time the routine is invoked.
- The simplest postcondition possible (and the default) is the expression *True*.
- This would mean we are happy no matter how the routine is invoked.

Compound preconditions

- Expressions can also be linked together using logical connectives like *and*, *or*, *not*, $=$, and *implies*.
- Writing two expressions on two consecutive lines without connectives is considered as an implicit *and*.
- It is also possible to invoke a separate *BOOLEAN* function containing complex computations.

Preconditions: invoking separate functions

```
my_feature (x: A_TYPE)
    require
        validity_check: is_valid_arg (x)
    do
        ...
    end
```

```
is_valid_arg (arg: A_TYPE): BOOLEAN
    do
        -- Complex computation here
    end
```

When things go bad . . .

The responsibility for a correct invocation lies on the *client*!

- An *exception* is raised whenever a client invokes a feature without obliging to its preconditions.
- The error should be relatively easy to fix, as *we know where in the code it occurred*.
- The *label* we put in the precondition comes in handy in this case.

Postconditions

Precondition: obligation for clients;

Postconditions

Precondition: obligation for clients;

Postcondition: benefit for clients.

Postconditions

Precondition: obligation for clients;

Postcondition: benefit for clients.

```
remove_all_segments
        -- Remove all stations except the first one.
    ensure
        only_one_left: count = 1
        both_ends_same: first = last
    end
```

Postcondition principle

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.

Postcondition principle

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 are postconditions in practice?

- Boolean expressions *checked after the execution* of the corresponding routine body.
- The simplest postcondition possible (and the default) is the expression *True*.
- This would mean we are happy no matter *what* the routine does.

Compound postconditions

- Expressions can also be linked together using logical connectives like *and*, *or*, *not*, $=$, and *implies*.
- Writing two expressions on two consecutive lines without connectives is considered as an implicit *and*.
- It is also possible to invoke a separate *BOOLEAN* function containing complex computations.

Postconditions: invoking separate functions

```
my_feature (x: A_TYPE)
    do
        ...
    ensure
        good_job_done: is_job_done_properly (x)
    end
```

```
is_job_done_properly (arg: A_TYPE): BOOLEAN
    do
        -- Complex computation here
    end
```

old notation

- Usable in *postconditions* only.
- It denotes value of an expression as it was on routine entry.
- eg. Bank Account

old notation

- Usable in *postconditions* only.
- It denotes value of an expression as it was on routine entry.
- eg. Bank Account

```
balance : INTEGER
        -- Current balance.

deposit (v: INTEGER)
        -- Add 'v' to account..
    require
        positive: v > 0
    do
        ...
    ensure
        added:
```

old notation

- Usable in *postconditions* only.
- It denotes value of an expression as it was on routine entry.
- eg. Bank Account

```
balance : INTEGER
          -- Current balance.

deposit (v: INTEGER)
          -- Add 'v' to account..
    require
        positive: v > 0
    do
        ...
    ensure
        added: balance = old balance + v
    end
```

When things go bad . . .

- If a client invokes a routine not satisfying its postcondition, an *exception* is raised.
- The error should be relatively *easy to fix*, as we know where in the code it occurred.
- The *label* we put in the postcondition comes in handy in this case.

The contract

Obligations	Benefits

The contract

	Obligations	Benefits
Client		

The contract

	Obligations	**Benefits**
Client	Satisfy the Precondition (Pay bill.)	

The contract

	Obligations	Benefits
Client	Satisfy the Precondition (Pay bill.)	From Postcondition (Receive telephone service from Supplier.)

The contract

	Obligations	**Benefits**
Client	Satisfy the Precondition (Pay bill.)	From Postcondition (Receive telephone service from Supplier.)
Supplier		

The contract

	Obligations	**Benefits**
Client	Satisfy the Precondition (Pay bill.)	From Postcondition (Receive telephone service from Supplier.)
Supplier	Satisfy Postcondition (Provide telephone service.)	

The contract

	Obligations	**Benefits**
Client	Satisfy the Precondition (Pay bill.)	From Postcondition (Receive telephone service from Supplier.)
Supplier	Satisfy Postcondition (Provide telephone service.)	From Precondition (No need to provide anything if bill not paid.)

Class invariants

Class invariants

Preconditions and postconditions are logical properties associated with a *particular feature*.

Class invariants

Preconditions and postconditions are logical properties associated with a *particular feature*.

Class invariants are properties that *all objects of a certain class must obey to*.

Class invariants

Preconditions and postconditions are logical properties associated with a *particular feature*.

Class invariants are properties that *all objects of a certain class must obey to*.

```
class DATE
...
invariant
    valid_day: 1 <= day and day <= 31
    valid_hour: 0 <= hour and hour <= 23
end
```

Class invariant principle

Class invariant principle

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

Class invariant principle

Class invariant principle

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

A class that fails to ensure its invariants is *buggy* software.

What are class invariants in practice?

- Boolean expressions checked *every time a supplier's status is observable by clients*:
  - As soon as an object is created;
  - Before and after a feature is available to clients.
- The simplest possible class invariant (and the default) is the expression *true*.
- This would mean that there are no consistency requirements.

Compound class invariants

- Expressions can also be linked together using logical connectives like *and*, *or*, *not*, $=$, and *implies*.
- Writing two expressions on two consecutive lines without connectives is considered as an implicit *and*.
- It is also possible to invoke a separate *BOOLEAN* function containing complex computations.

When things go bad . . .

- An *exception* is raised if an invariant check fails:
  - Immediately after an object is *created*;
  - Before or after a client invokes a routine.
- The error should be relatively *easy to fix*, as we know where in the code it occurred.
- The *label* on the class invariant can be useful.

When breaking class invariant is OK

- A class invariant *is allowed to be not satisfied* in the following cases:
  - *Before* the invocation of a *creation* feature;
  - *In the body* of any routine.
- Anything can happen *between states observable by clients*

Why Design by Contracts!

Applications of Design by Contract

- More efficient debugging:
  - programs *fail earlier* $\Rightarrow$ cheaper to fix;
  - it is easier to reason about errors because they are *better localised*.
- Better *documentation*:
  - contracts are part of the *class interface*;
  - clients know *how to invoke* a feature correctly and *what* to expect in return;
  - part of the documentation is *executable*.

Applications of Design by Contract (2)

- Automatic testing:
  - tests can be generated automatically from contracted code;
  - *preconditions* filter out uninteresting/wrong inputs;
  - *postconditions* are oracles, providing a pass/fail response
- Theorem provers
  - it is possible to use assertions to prove that certain properties hold in a program.

Is Design by Contract a silver bullet?

Is Design by Contract a silver bullet?

- No

Is Design by Contract a silver bullet?

- No, but it may help to produce better software.

Is Design by Contract a silver bullet?

- No, but it may help to produce better software.

In the end contracts are written by humans

Is Design by Contract a silver bullet?

- No, but it may help to produce better software.

In the end contracts are written by humans:Humans can write wrong, or weak contracts.

Think about what happens if you always write *True* in all your contract clauses.

What we have seen today

- Classes
- The notion of interface
- GUI vs API
- Contracts
- Preconditions
- Postconditions
- Class invariants

Reading Assignment: Chapter 4 of "Touch of Class, Learning to Program Well with Objects and Contracts"