Design by contract

Motivations

- formal specification of classes and object interfaces to guarantee correctness of software
- two parties are involved in the contract:
  - the developers of a class *C*
  - the clients of *C* (=programmers that use *C* in their programs)
- the contract in a nutshell: if clients use a class *C* correctly then
  - method calls on *C* objects behave in accordance with their specification
  - the state of any *C* object is always valid after
    - its creation
    - any method call on it

Formalization

- use a class correctly = pre-conditions
- method calls behave in accordance = post-conditions
- the state of any *C* object is always valid = invariants

Design by contract

Code contracts: pre-conditions, post-conditions, invariants

- pre-condition for method *m*, defined by a predicate *p*
  `requires` *p*: *p* must hold immediately before the execution of *m*
- post-condition for method *m*, defined by a predicate *p*
  `ensures` *p*: if the pre-condition of *m* holds, then *p* must hold immediately after the execution of *m*
- invariant for class *C*, defined by a predicate *p*
  `invariant` *p*: *p* must hold
  - immediately after creation of each instance of *C*;
  - immediately before the execution of each instance method of *C*;
  - immediately after the execution of each instance method of *C*, if the pre-condition of the method holds.

Design by contract

Syntactic entities

- standard logical connectives and quantifiers
- pre-conditions: the parameters of methods, `this` and object fields
- post-conditions: the parameters and the result of methods, `this` and the old (before the call) and new (after the call) values of the object fields
- invariants: object fields

Remarks

- Java does not offer native support for design by contract
- other languages do (Eiffel)
- pre/post-conditions and invariants in Java comments are useful

Design by contract

Example

```
public class TimerClass {
    private int time;
    /* invariant 0 <= time && time <= 3600; */
   public int getTime() {
    /* ensures result == this.time && this.time == old(this.time); */
      return this.time;
   }
    public boolean isRunning() {
    /* ensures result == this.time > 0 && this.time == old(this.time); */
      return this.getTime() > 0;
    }
    public int reset(int minutes) {
    /*  requires 0 <= minutes && minutes <= 60;
        ensures  result == old(this.time)
                 && this.time == minutes * 60; */
      if (minutes < 0 || minutes > 60)
         throw new IllegalArgumentException();
      int prevTime = this.getTime();
      this.time = minutes * 60;
      return prevTime;
    }
    ...
}
```

Class invariants

How class invariants can be ensured to hold?

- information hiding: object states changed in a controlled way only with methods, no arbitrary changes allowed!
- all methods preserve invariants
- initially, the invariant must be verified by construction
- constructors are used for initializing objects correctly, while guaranteeing information hiding
- it is not possible to create a new object without initializing it:
  a constructor will be always called

Constructors

Example of definition

`TimerClass` with three different constructors:

```
public class TimerClass {
    private int time;

    public TimerClass() {                  // 'time' has the default value 0
    }
    public TimerClass(int minutes) {       // 'time' has value minutes * 60
        if (minutes < 0 || minutes > 60)
            throw new IllegalArgumentException();
        this.time = minutes * 60;
    }
    public TimerClass(TimerClass other) { // copy constructor
        this.time = other.getTime();
    }
...
}
```

Remark

A class can have multiple constructors
Terminology: constructors can be overloaded

Constructors

Demo

```
TimerClass t1 = new TimerClass();
TimerClass t2 = new TimerClass(42);
TimerClass t3 = new TimerClass(t2);
assert t1.getTime()==0 && t2.getTime()==42*60 &&
       t2.getTime()==t3.getTime();
```

Field initializers

Another way to initialize objects

Object can be also initialized with field initializers

Example of variable initializer

```
public class TimerClass {
   private int time = 60; // default value for ’time’ is 60 seconds

    public TimerClass() { // keeps the default value of ’time’
    }
    public TimerClass(int minutes) {
        if (minutes < 0 || minutes > 60)
            throw new IllegalArgumentException();
        this.time = minutes * 60;
    }
    public TimerClass(TimerClass otherTimer) {
        this.time = otherTimer.getTime();
    }
...
}
```

Object creation and initialization

Simplified rules

1. immediately after object creation a default value is assigned to each field of the object
2. the default value depends on the type of the field:
    - 0 for `int` and other numerical types
    - `false` for `boolean`
    - `null` for reference types
3. field initializers are executed in the left-to-right top-to-bottom order
4. a constructor of the class is called, according to the number and types of arguments

Object creation and initialization

Demo

```
TimerClass timer1 = new TimerClass();
TimerClass timer2 = new TimerClass(1);
assert timer1.getTime() == timer2.getTime();
```

Rules on constructors

Overloaded constructors

- multiple constructors can be defined in the same class
- they must be distinguishable by
  - numbers of parameters
  - types of parameters

Default constructor

- if no constructor is declared, then a default one is added to the class
- the default constructor has no parameters and its body is empty

Explicit constructor call

Demo

```
public class Person {
    private String name;
    /* invariant name != null */

    private String address;

    public Person(String name) {
        if (name == null)
            throw new NullPointerException();
        this.name = name;
    }
    public Person(String name, String address) {
        this(name); // calls the first constructor
        this.address = address;
    }
    public String getName() {
        return this.name;
    }
    public String getAddress() {
        return this.address;
    }
}
```

Explicit constructor call

Rules

- a constructor may be explicitly called in another constructor
- syntax: `'this' '(' (Exp ( ',' Exp)*)? ')'`
- explicit call allowed only on the first line of a constructor
- cyclic constructor calls not allowed

Java convention for constructor parameters

- parameters have the same name of the corresponding fields
- example:

```
public Person(String name, String address) {
    ...
}
```

  - parameter `name` is used to initialize **`this`**`.name`
  - parameter `address` is used to initialize **`this`**`.address`

Explicit constructor call

Demo

```
Person sam = new Person("Samuele");
Person sim = new Person("Simone","Genova");
assert sam.getAddress()==null && sim.getAddress()!=null;
```

Remarks

Object fields in statically typed languages

- fields cannot be added to or removed from objects
- solution:
  - all fields of the object must be declared in the class
  - `null` is used to indicate that a field is unsed
- Remark: `null` cannot be used for primitive types `int`, `boolean`
- for primitive types `int`, `boolean` there is no value to indicate that the field is unused

Java strings in a nutshell

- `String` is a predefined class
- strings are immutable objects
- string literals have a standard syntax; example: `"Genova"`