Conception orientée objet

DUT M3105 : Conception et programmation objet avancées

Serge Rosmorduc
serge.rosmorduc@lecnam.net
Conservatoire National des Arts et Métiers

2018-2019

Organisation du cours

- Cours + séance de TP ;
- Le TP doit être déposé sous GIT ;
- Vérifiez bien, à la fin de la séance, que vous avez déposé le TP dans son état actuel ;
- Chaque TP a une date de rendu final (toujours sous GIT) ;
- *Contrôles surprise possible en début de cours.* Généralement au sujet d'un travail qu'on vous aura demandé de terminer.
- devoir sur table en fin de séquence.

Classe et objet

- Un objet *représente* une chose, réelle ou imaginaire, concrète ou abstraite, en permettant de la manipuler à travers des propriétés et des méthodes.
- Les objets qui ont le même comportement sont regroupés (et décrits) à travers la même *classe.*
- La classe `Voiture` décrit ce qu'est *une* voiture en général, en précisant ses propriétés et son comportement. L'objet `vehiculeJoueur`, représente une voiture en particulier, et est une *instance* de la classe `Voiture`.
- Noter une différence entre le concept de voiture en général, et la classe `Voiture` de telle ou telle application. On ne travaille jamais qu'avec un type d'utilisation spécifique des objets en vue.
- En pratique, le nom d'une classe est un *substantif ;* les méthodes sont plutôt des *verbes.*

Comment délimiter une classe

- une classe représente un concept et un seul, et le représente bien ;
- on doit pouvoir dire ce que *représente* un objet de la classe ;
- si l'explication et longue et alambiquée, remplacer par *plusieurs* classes ?

Objectifs du cours

- Vous aider à trouver les bonnes classes pour *modéliser* un problème ;
- vous aider à les *structurer* entre elles ;
- vous aider à leur attribuer des *responsabilités* dans vos programmes : qui crée quoi ? par exemple ;
- vous fournir des outils d'analyse ;
- des exemples d'organisation ;
- un vocabulaire pour décrire tout ça (les patterns) ;

La programmation par contrats

Définie par Bertrand Meyer (Object Oriented Software Construction).

- Idée : considérer qu'une classe « passe » un contrat avec le code qui l'utilise (le code client) ;
- ce contrat est composé :
  - de pré-conditions : conditions logiques qui doivent être vraies pour que le client ait le droit d'appeler une méthode donnée.

Pour un objet CompteBanquaire, la méthode retirer(somme) peut avoir comme précondition que somme soit positif et inférieur à solde()

  - pos-condition : condition qui est vraie après l'appel si la pré-condition est vérifiée.

pour retirer : nouveau solde() = ancien solde() - somme.

  - les conditions s'expriment *indépendemment* de l'implémentation de la classe.

Notion d'invariants

- un invariant est une condition qui est vérifiée par un objet avant et après chaque appel de méthode ;
- exemples d'invariants :

Pour un compte banquaire : $solde() \geq 0$

Pour une liste triée : $\forall i, i \in 1..l.size() - 1 \rightarrow l.get(i) \geq l.get(i-1)$

- les invariants sont nécessaires ;
- ils sont utiles pour raisonner : quand j'insère un nouvel élément dans une liste triée, je *sais* que celle-ci est triée.

Constructeurs, méthodes et invariants

- tout constructeur doit établir l'invariant de la classe ;
- si l'invariant est vrai avant un appel de méthode, il est vrai après ;
- donc, l'invariant est *toujours* vrai.

Remarque sur les pré-conditions

- D'après B. Meyer (OOSC, 2ed. p. 342), *les méthodes ne sont pas responsable du test de leurs pré-condition.*
- On peut utiliser les assertions du langage java (`assert`) ;
- On peut transformer une précondition en spécification « normale » du programme :

Précondition

```
/**
 * Retourne le maximum de t.
 * Pré-condition: t est non null et
 * de taille supérieure à 0.
 * Post-condition retourne x tel que x dans t
 *   et x supérieur ou égal à tout élément de t.
 */
public int max(int t[]) {...}
```

Dans cette version, le comportement de `max` **n'est pas défini** si la précondition n'est pas vérifiée. C'est au client de faire attention.

Remarque sur les pré-conditions

Levée d'exception

```
/**
 * Retourne le maximum de t.
 *
 * Post-condition retourne x tel que x dans t
 *   et x supérieur ou égal à tout élément de t,
 *   ou lève une IllegalArgumentException si t
 *   est null ou de taille 0.
 */
public int max(int t[]) {...}
```

Ici, on n'a plus de pré-condition, mais la méthode indique explicitement comment elle traite les cas incorrects.

Les tests

- test unitaires : vérifie le fonctionnement de chaque méthode individuellement, en isolant les objets le plus possible ;
- tests d'intégration : vérifient que plusieurs objets collaborent correctement ensemble, sur un scenario prédéfini.
- test de non-régression : on conserve les anciens tests ; quand le logiciel évolue, une erreur sur un test qui passait auparavant est une régression.

Test Driven Development

- On définit l'en-tête des méthodes, mais pas leur implémentation ;
- on écrit les tests à réaliser ;
- on implémente et on teste ;
- on corrige les erreurs du code...
- jusqu'à ce qu'il tourne.

JUnit

- framework de test simple (mais efficace) ;
- créé par
  - Kent Beck (Xtreme programming)
  - Erich Gamma (Design Patterns)
- permet d'écrire et de réaliser des tests en java simplement ;
- beaucoup de dérivés pour d'autres langages (phpunit, par exemple) ;
- http ://junit.sourceforge.net/doc/cookbook/cookbook.htm
- Thomas & al, Java testing patterns, Wiley 2004
- Marc Clifton,
  http ://www.codeproject.com/KB/architecture/autp5.aspx

Exemple

```
public class CompteTest {
  @Test
  public void testeDepot() {
     Compte c= new Compte();
     c.deposer(100);
     Assert.assertEquals(100, c.getSolde());
  }

  ...
}
```

Exemple : un ensemble trié

Soit une classe `EnsembleEntiersTries`, dotée des méthodes

- size() : nombre d'éléments dans l'ensemble ;
- int getElement(int i) : élément en position i ;
- boolean contains(int v) : vrai ssi l'ensemble contient v ;
- isEmpty() : vrai ssi l'ensemble est vide ;
- void addElement(int v) : ajoute v à l'ensemble ;
- void removeElement(int v) : supprime v de l'ensemble ;

Contrat

- **Invariant de classe** (valide avant et après tout appel de méthode) :

les éléments sont triés et uniques

$$\forall i, j\,(0 \leq i < j < s.size()) \rightarrow s.get(i) < s.get(j)$$

- Contrat pour s.addElement(v) :

post-condition : s.contains(v) est vraie ;

- Contrat pour s.contains(v) :

Post-condition :

$$s.contains(v) = true \Leftrightarrow \exists i, i \in [0..s.size() - 1] \wedge s.get(i) = v$$

Utilisation pour le test

Les contrats indiquent *quoi* tester.

```
@Test
  public void testAddToEmpty() {
      final int val = 3443;
      testSet= new SimpleIntSet(); // ensemble vide
      assertFalse(testSet.contains(val));
      testSet.addElement(val); // ajoute l'élément
      assertTrue(testSet.contains(val));
  }
```

En pratique

- Les tests sont conservés dans des classes à part (dans un dossier `test`) ;
- chaque test est écrit dans une méthode, annotée avec `@Test` ;
- on utilise les méthodes de la classe `org.junit.Assert` pour vérifier que les tests sont corrects ;
- *en théorie*, une méthode de test fait un seul test ;

Exemple très simple

Testons une classe qui implémente cette interface :

```
public interface DivisionComputer {
    /**
     * Asks for the computation of p/q.
     * @param p
     * @param q
     */
    public void divide(int p, int q);
    /**
     * Returns the quotient part of the last operation.
     * If the divisor was 0, throws ArithmeticException.
     */
    public int getQuotient();
    /**
     * Returns the remainder part of the last operation.
     * If the divisor was 0, throws ArithmeticException.
     */
    public int getRemainder();
}
```

Implémentation à tester

```
package cnam.utils;
public class DivisionComputerImplementation
                    implements DivisionComputer {
    private int quotient, remainder;
    private boolean defined;
    public int getQuotient() {
        if (defined)
            return quotient;
        else
            throw new ArithmeticException("/ by 0");
    }
    public int getRemainder() {
        if (defined)
            return remainder;
        else
            throw new ArithmeticException("/ by 0");
    }
```

```java
public void divide(int p, int q) {
        defined = q != 0;
        if (defined) {
            int sign = 1;
            if ((p < 0 && q > 0) || (p > 0 && q < 0)) {
                sign = -1;
            }
            remainder = Math.abs(p);
            int denominator = Math.abs(q);
            quotient = 0;
            while (remainder > denominator) {
                remainder = remainder - denominator;
                quotient++;
            }
            quotient= quotient * sign;
            // To keep the relation p= q*quotient + remaind
            // remainder should have the same sign as p :
            if (p < 0)
                remainder= - remainder;
        }
    }
}
```

Une classe de test

```
package cnam.utils;

import org.junit.Test;
import static org.junit.Assert.*;

public class DivisionComputerImplementationTest {
    @Test
    public void testQuotient() {
        DivisionComputer c = new
            DivisionComputerImplementation();
        c.divide(30, 4);
        assertEquals(7,c.getQuotient());
        assertEquals(2,c.getRemainder());
    }

... other tests ...
```

(notez l'import : les assertions se trouvent dans org.junit.Assert).

assertEquals

`public static void assertEquals(long expected, long actual)`
vérifie que deux long sont égaux. S'ils ne le sont pas, une exception `AssertionError` *est levée.*
Paramètres :

- expected : valeur attendue ;
- actual : valeur constatée ;

`public static void assertEquals(java.lang.String message, longexpected, long actual)`
vérifie que deux long sont égaux. S'ils ne le sont pas, une exception `AssertionError` *est levée, accompagnée du message passé en premier argument.*

Autres méthodes

- assertArrayEquals : compare arrays content
- assertTrue
- assertFalse
- assertNotNull
- assertSame assert two references have the same address
- assertThat general assert (using the Matcher class).

Tester les exceptions

```
@Test(expected = ArithmeticException.class)
 public void divideByZeroB() {
    DivisionComputer c = new DivisionComputerImplementa
    c.divide(100,0);
    int q = c.getRemainder();
}
```

Tester les boucles infinies

ça n'est pas possible, mais on peut donner un temps maximal d'exécution aux tests :

- pour un test particulier :

```
1     @Test(timeout = 1000)
2     public void testB() {
3        ...
4     }
```

- pour tous les tests d'une classe :

```
1  public class DemoTest {
2  @Rule
3  public TestRule timeout = Timeout.seconds(2);
4  ...
5  }
```

```java
package cnam.utils;

import org.junit.Test;
import static org.junit.Assert.*;

public class DivisionComputerImplementationTest {

    @Test
    public void testQuotient() {
        DivisionComputer c = new DivisionComputerImplementa
        c.divide(30, 4);
        assertEquals(7,c.getQuotient());
        assertEquals(2,c.getRemainder());
    }
    @Test
    public void testDivisionByOne() {
        DivisionComputer c = new DivisionComputerImplementa
        c.divide(30, 1);
        assertEquals(30,c.getQuotient());
        assertEquals(0,c.getRemainder());
    }
```

```java
@Test
public void testExactDivision() {
    DivisionComputer c = new DivisionComputerImplementa
    c.divide(15, 5);
    assertEquals(3,c.getQuotient());
    assertEquals(0,c.getRemainder());
}

@Test
 public void divideMinusByPlus() {
    DivisionComputer c = new DivisionComputerImplementa
    c.divide(-30, 4);
    assertEquals(-7,c.getQuotient());
    assertEquals(-2,c.getRemainder());
}
```

```
    @Test
     public void divideMinusByMinus() {
        DivisionComputer c = new
DivisionComputerImplementation();
        c.divide(-30, -4);
        assertEquals(c.getQuotient(), 7);
        assertEquals(c.getRemainder(), -2);
    }
@Test
    public void testDivideZero() {
        DivisionComputer c = new DivisionComputerImplementa
        c.divide(0, 100);
        assertEquals(0,c.getQuotient());
        assertEquals(0,c.getRemainder());
    }
```

```java
@Test(expected = ArithmeticException.class)
public void divideByZeroA() {
    DivisionComputer c = new DivisionComputerImplementa
    c.divide(100,0);
    int q = c.getQuotient();
}

@Test(expected = ArithmeticException.class)
 public void divideByZeroB() {
    DivisionComputer c = new DivisionComputerImplementa
    c.divide(100,0);
    int q = c.getRemainder();
}
```

```
@Test
 public void dividePlusByMinus() {
    DivisionComputer c = new DivisionComputerImplementa
    c.divide(30, -4);
    assertEquals(-7,c.getQuotient());
    assertEquals(2, c.getRemainder());
}
```

Lancer les tests

- en mode ligne :

```
1  java org.junit.runner.JUnitCore my.Test
```

- en général :
  - dans l'IDE (eclipse/netbeans) ;
  - en utilisant le système de build (ant/maven) : automatique à la compilation avec maven ;
  - dans un environnement d'intégration continue (hudson) ;

Concevoir ses tests

- si elles ne sont pas disponibles, écrire le plus précisément possible les spécifications des méthodes ;
- l'écriture des tests amène aussi à préciser les spécifications ;
- pour chaque méthode, *en partant de la spécification* :
  - tester les cas « normaux » ;
  - chercher les cas limites (typiquement : bornes pour une application numérique, listes vides pour les listes) et les tester ;
  - dans les tests, vérifier qu'on obtient le résultat attendu, que les post-conditions sont vérifiées, et que les invariants de classe sont bien respectés ;

Test et bug

Que faire quand on rencontre un bug dans un programme ?

1. écrire un test pour mettre le bug en évidence ; expliquer en commentaire le pourquoi de ce test ;
2. corriger le bug ;
3. vérifier que le test passe maintenant correctement ;