

# Design Patterns

©Sovanna Tan  
sovanna.tan@u-pec.fr

Septembre 2015

# Organisation du cours

- 30h Cours/TD/TP (java 1.8 avec eclipse)
- Projet commun avec Java avancé

# Contenu du cours

- Bonnes pratiques de génie logiciel
- Introduction aux techniques de test logiciel
- Programmation objet avec du code de qualité
  - Modifiable
  - Réutilisable
- **Design Patterns**, en français, patrons ou modèles de conception : des solutions objet qui ont fait leurs preuves à des problèmes courants

# Bibliographie

- *Design Patterns en Java. Les 23 modèles de conception des-criptions et solutions illustrées en UML2 et Java*, LAURENT DEBRAUWER, Éditions ENI, 2013.
- *Design Patterns. Catalogue de modèles de conception réutilisables*, ERICH GAMMA, RICHARD ELM, RALPH JOHNSON, JOHN VLISSIDES, Vuibert, 2007.
- *Head First Design Patterns*, ERIC FREEMAN, ELISABETH ROB-SON, O'Reilly, 2014.
- *The Art of Software Testing*, GLENFORD J. MYERS, COREY SANDLER, TOM BADGETT, John Wiley & Sons, 2012.
- *Pragmatic Unit Testing in Java 8 with JUnit*, JEFF LANGR, The pragmatic Programmers, 2015.
- *Coder proprement*, ROBERT C. MARTIN, Pearson France, 2013.
- *UML 2 de l'apprentissage à la pratique*, LAURENT AUDIBERT, Ellipses, 2009.

# Plan

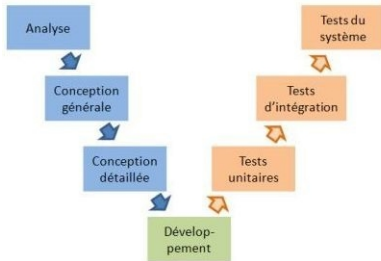
- 1 Introduction
- 2 Coder proprement
- 3 Le test logiciel
- 4 Les diagrammes de classes UML
- 5 Design Patterns

- 1 Introduction
- 2 **Coder proprement**
- 3 Le test logiciel
- 4 Les diagrammes de classes UML
- 5 Design Patterns

# Le développement

## Cycle en V

Calendrier global du projet sur une longue période



## Méthode Agile

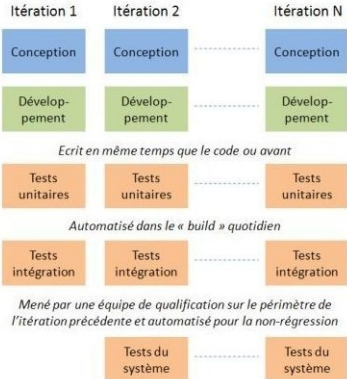


Image provenant de [http://tisserant.org/cours/qualite-logiciel/qualite\\_logiciel.html](http://tisserant.org/cours/qualite-logiciel/qualite_logiciel.html)

# Bien choisir les noms

- Classe : nom
- Fonction : verbe

```
int d; // elapsed time in days
```

- Le nom doit indiquer le contenu de la variable.
- Le code doit se passer de commentaires.
- Le nom doit être facile à trouver dans le fichier source.

```
int elapsedTimeInDays;  
int daysSinceCreation;  
int daysSinceModification;  
int fileAgeInDays;
```

- Les distinctions doivent être significatives : préférer **source** et **destination** à **a1** et **a2**.



# Le code contient les intentions

```
public List<int[]> getThem() {  
    List<int[]> list1 = new ArrayList<int[]>();  
    for (int[] x : theList)  
        if (x[0] == 4)  
            list1.add(x);  
    return list1;  
}
```

- Version tableau d'entiers améliorée

```
public List<int[]> getFlaggedCells() {  
    List<int[]> flaggedCells = new ArrayList<int[]>();  
    for (int[] cell : gameBoard)  
        if (cell[STATUS_VALUE] == FLAGGED)  
            flaggedCells.add(cell);  
    return flaggedCells;  
}
```

- Version objet

```
public List<Cell> getFlaggedCells() {  
    List<Cell> flaggedCells = new ArrayList<Cell>();  
    for (Cell cell : gameBoard)  
        if (cell.isFlagged())  
            flaggedCells.add(cell);  
    return flaggedCells;  
}
```

# Open Close Principle

Une classe en production doit être

- fermée à la modification,
- ouverte à l'extension : on peut ajouter des fonctionnalités en gardant le code existant.

Utiliser

- l'abstraction,
- le polymorphisme.

# Open Close Principle : mauvais exemple

```
class GraphicEditor {  
  
    public void drawShape(Shape s) {  
        if (s.m_type==1)  
            drawRectangle(s);  
        else if (s.m_type==2)  
            drawCircle(s);  
        }  
    public void drawCircle(Circle r) {...}  
    public void drawRectangle(Rectangle r) {...}  
}  
  
class Shape {  
    int m_type;  
}  
  
class Rectangle extends Shape {  
    Rectangle() {  
        super.m_type=1;  
    }  
}  
  
class Circle extends Shape {  
    Circle() {  
        super.m_type=2;  
    }  
}
```

# Open Close Principle : exemple amélioré

```
class GraphicEditor {  
    public void drawShape(Shape s) {  
        s.draw();  
    }  
}
```

```
class Shape {  
    abstract void draw();  
}
```

```
class Rectangle extends Shape {  
    public void draw() {  
        // draw the rectangle  
    }  
}
```

# Fonctions

## Éviter les arguments de sortie

Une fonction qui modifie un de ses paramètres doit être une méthode de la classe du paramètre modifié.

## Stepdown rule

Dans un fichier les fonctions apparaissent dans l'ordre de la plus abstraite à la moins abstraite.

## Selon R.C. Martin, une fonction doit

- ne faire qu'une seule chose,
  - être courte,
  - avoir peu d'arguments,
  - avoir un seul niveau d'abstraction, le même niveau de détails.
- 
- facile à tester, adaptée au Test Driven Development (TDD)

# Ne pas écrire<sup>1</sup>

```
public void add(Object element) {  
    if (!readOnly) {  
        int newSize = size + 1;  
        if (newSize > elements.length) {  
            Object[] newElements =  
                new Object[elements.length + 10];  
            for (int i = 0; i < size; i++)  
                newElements[i] = elements[i];  
            elements = newElements;  
        }  
        elements[size++] = element;  
    }  
}
```

1. image provenant de <http://programmers.stackexchange.com/questions/195989/>

# Mais plutôt <sup>2</sup>

```
public void add(Object element) {  
    if (readOnly)  
        return;  
    if (atCapacity())  
        grow();  
    addElement(element);  
}
```

---

2. image provenant de <http://programmers.stackexchange.com/questions/195989/>

is-it-ok-to-split-long-functions-and-methods-into-smaller-ones-even-though-they

- 1 Introduction
- 2 Coder proprement
- 3 Le test logiciel**
- 4 Les diagrammes de classes UML
- 5 Design Patterns



# Le test logiciel

## Vérification partielle du logiciel

Assurance relative du bon fonctionnement : le logiciel réagit comme prévu par les concepteurs et est conforme aux attentes du client.

- Vérification tout au long du développement
- Conformité du logiciel à sa spécification
- Implantation correcte de la conception
- Qualité du logiciel (fiabilité, robustesse, sécurité, portabilité, ...)
- Réduction des coûts

# Définitions

## IEEE (Institute of Electrical and Electronics Engineers)

Le test est l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus.

## AFCIQ (Association Française pour le Contrôle Industriel et la Qualité)

Le test est une technique de contrôle consistant à s'assurer, au moyen de son exécution, que le comportement d'un programme est conforme à des données préétablies.

# Citations

G. J. MYERS (The Art of Software Testing)

Testing is the process of executing a program with the intent of finding errors.

E. W. DIJKSTRA (Notes on structured programming)

Testing can reveal the presence of errors but never their absence.

# Vocabulaire

Error, mistake

Erreur de programmation

Bug, Defect

Manifestation de l'erreur dans le code

Anomaly, Failure

Différence entre le comportement observé et le comportement attendu

- Erreur  $\Rightarrow$  Bug  $\Rightarrow$  Défaillance

# Visibilité

## Test en boîte noire

Test fonctionnel à partir de l'analyse de la spécification du logiciel

- Détection des erreurs d'omission, d'initialisation, de terminaison, d'interface, de structure de données.

## Test en boîte blanche

Test structurel à partir de l'analyse de l'implémentation du logiciel

- Détection des erreurs de programmation dans les structures de contrôle (condition, boucle), la gestion de la mémoire.

# Test statique/dynamique

## Test statique

- Revue de code : relecture par des pairs
- Analyse statique automatique avec des outils logiciels

## Test dynamique

- Exécution dynamique du logiciel sur un certain nombre de données de test
- Vérification des résultats : valeurs en sortie, comportement du système, ...

## Oracle

Moyen d'évaluation des résultats d'un test. Il peut être automatique, semi-automatique ou manuel.

# Dans le cycle de développement

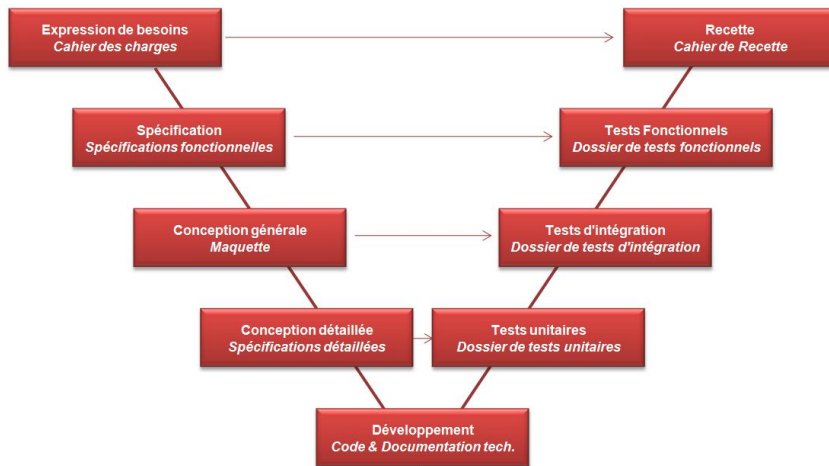


Image provenant de <http://www.aggil.com/>

# Les différents niveaux

## Test unitaire

Procédure pour s'assurer du bon fonctionnement d'une unité de programme.

## Test d'intégration

Test du fonctionnement lorsque l'on regroupe les différentes unités de programme. Vérifier que les composants communiquent entre eux correctement.

## Test fonctionnel

Test en boîte noire du logiciel à partir des spécifications.

## Test système

Vérification que le système fonctionne du point de vue de l'utilisateur.



# Les tests de recette

## Recette

Vérification de conformité du produit fonctionnelle et technique devant conduire à la mise en production du logiciel par le client.

## Test de performance

Analyse des performances avec différents niveaux de charge.

## Test aux limites

Étudier le comportement dans des conditions extrêmes.

## Test de montée en charge

Analyse des performances quand on augmente la charge.

## Test d'endurance

Analyse des performances en charge pendant une longue durée.

# Maintenance

## Test de non régression

Vérification que les mises à jour ne cassent pas les fonctionnalités.

- Les tests unitaires œuvrent dans ce sens.

# Assert en java

```
assert booleanExpression ;
```

- Si `booleanExpression` vaut **false**, une exception `AssertionError` est levée.

```
assert booleanExpression : errorMessage ;
```

- Si `booleanExpression` vaut **false**, `errorMessage` est évalué (il ne doit pas valoir **void**) et est passé en argument au constructeur de `AssertionError` en tant que message d'erreur détaillé.
- On active les assertions à l'exécution avec `java -ea`.

# Tests unitaires<sup>3</sup>

Code exécuté pour vérifier la validité d'une unité de programme.

- Vérification du comportement sur des entrées correctes
- Vérification du comportement sur des entrées incorrectes

---

3. Partie du cours inspirée par <http://www.emse.fr/~picard/cours/2A/junit/junit.pdf>

# Les tests unitaires en java

## JUnit 4

Framework pour les tests unitaires en java développé par

- KENT BECK (eXtreme Programming qui utilise le Test Driven Development)
- ERICH GAMMA (Design Patterns)
- <http://junit.org/>
- plugin pour eclipse

## Fonctionnement

- Une classe de tests unitaires est associée à la classe à tester.
- On utilise les annotations java pour repérer les méthodes.
- On utilise des assertions pour faire les tests.

# Les assertions

<code>fail(String message)</code>	fait échouer le test
<code>assertTrue(boolean condition)</code>	vrai si condition vaut vrai
<code>assertEquals(expected, actual)</code>	vrai si les valeurs sont égales
<code>assertEquals(expected, actual, tolerance)</code>	vrai si les valeurs sont proches
<code>assertNull(object)</code>	vrai si <b>null</b>
<code>assertNotNull(object)</code>	vrai si différent de <b>null</b>
<code>assertSame(expected, actual)</code>	vrai si référencent le même objet
<code>assertNotSame(expected, actual)</code>	vrai si ne référencent pas le même objet

# Test

## Méthode de test

- Visibilité **public**
- Type de retour **void**
- Ne prend pas de paramètre
- Peut lever une exception
- Annotée @Test
- Utilise les assertions

# Les annotations

@Test	méthode de test
@Before	méthode exécutée avant chaque test
@After	méthode exécutée après chaque test
@BeforeClass	méthode exécutée avant le premier test
@AfterClass	méthode exécutée après le dernier test
@Ignore	méthode qui n'est pas exécutée comme test



# Exemple classe à tester

```
package fr.uec.test;

public class Money {
    private int fAmount;
    private String fCurrency;
    public Money(int amount, String currency) {
        fAmount = amount;
        fCurrency = currency;
    }
    public int amount() {
        return fAmount;
    }
    public String currency() {
        return fCurrency;
    }
    public Money add(Money m) {
        return new Money(amount() + m.amount(), currency());
    }
    public boolean equals(Money m){
        return (fAmount==m.amount() && fCurrency.equals(m.currency()));
    }
}
```

# Exemple classe de test

```
package fr.upec.test;

import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;
import org.junit.Before;
import org.junit.Test;

public class MoneyTest {
    private Money f12CHF;
    private Money f14CHF;
    private Money f26CHF;

    @Before
    public void setUp() {
        f12CHF= new Money(12, "CHF");
        f14CHF= new Money(14, "CHF");
        f26CHF= new Money(26, "CHF");
    }

    @Test
    public void testMoney(){
        assertNotNull(f12CHF.currency());
        assertNotNull(f12CHF);
    }

    @Test
    public void testAdd() {
        Money result = f12CHF.add(f14CHF);
        assertTrue(f26CHF.equals(result));
    }
}
```

# Les bonnes pratiques<sup>4</sup>

- Écrire les tests en même temps que le code.
- Tester seulement ce qui peut provoquer des erreurs.
- Exécuter les tests à chaque modification du code.
- Écrire un test pour tout bug trouvé.
- Ne pas tester plusieurs méthodes dans un même test : JUnit s'arrête à la première erreur.

---

4. inspiré par [http://www.liafa.jussieu.fr/~sighirea/cours/methtest/c\\_JUnit.pdf](http://www.liafa.jussieu.fr/~sighirea/cours/methtest/c_JUnit.pdf)

- 1 Introduction
- 2 Coder proprement
- 3 Le test logiciel
- 4 Les diagrammes de classes UML**
- 5 Design Patterns

- 1 Introduction
- 2 Coder proprement
- 3 Le test logiciel
- 4 Les diagrammes de classes UML
- 5 Design Patterns**