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 descriptions 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 Robson, 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

Le développement

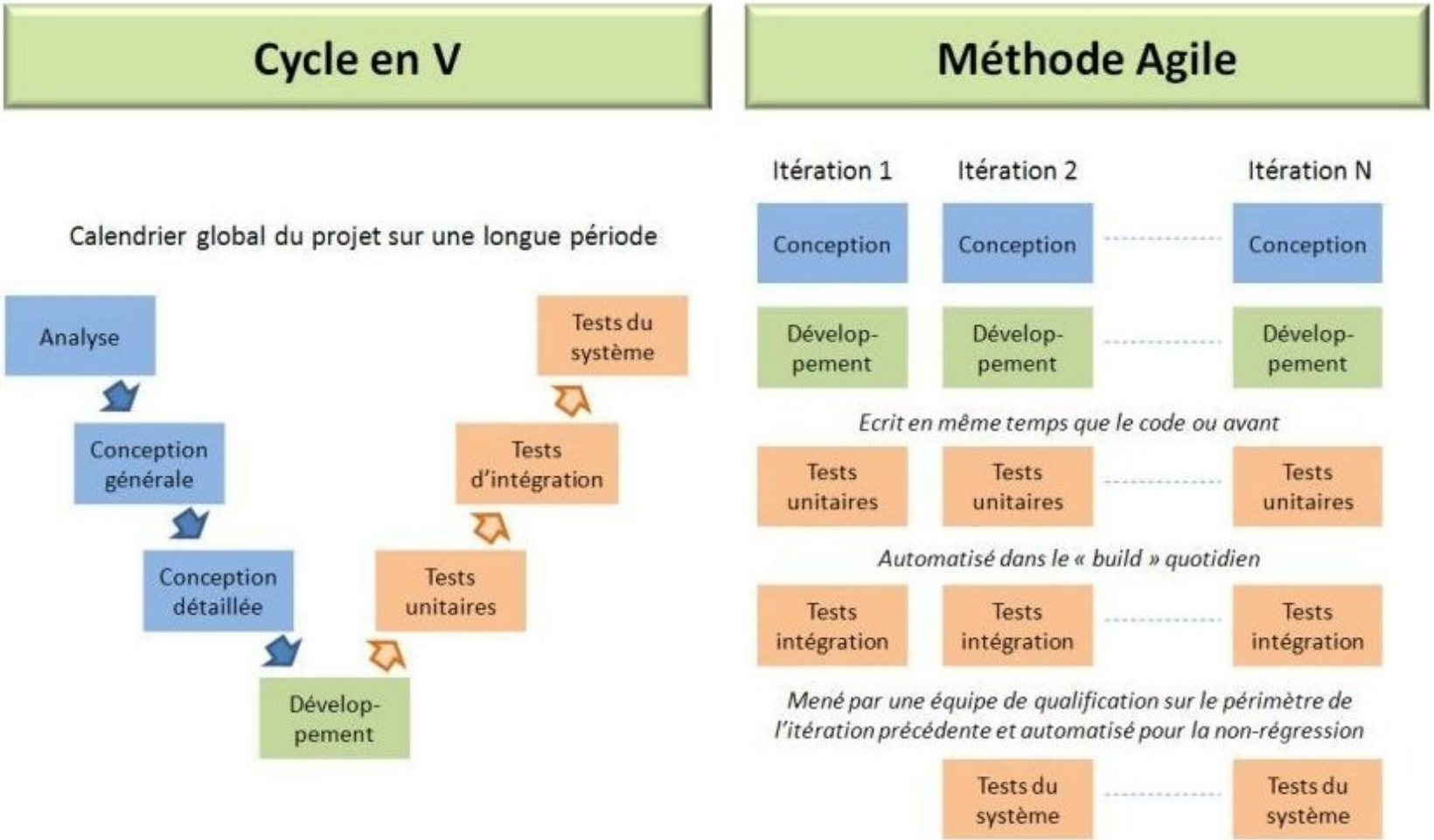


Image provenant de `http://tisserant.org/cours/qualite-logiciel/qualite_logiciel.html`

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;
}
```

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 [1]

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

Mais plutôt [2]

```
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

Single Responsibility Principle (SRP)

Une classe, une responsabilité

- Si une responsabilité change, seule la classe en charge est modifiée.

```
// single responsibility principle - bad example
interface IEmail {
   public void setSender(String sender);
   public void setReceiver(String receiver);
   public void setContent(String content);
}

class Email implements IEmail {
   public void setSender(String sender) { ... }
   public void setReceiver(String receiver) { ... }
   public void setContent(String content) { ... }
}
```

Exemple tiré de `http://www.oodesign.com/single-responsibility-principle.html`

Version respectant le SRP

```
// single responsibility principle - good example
interface IEmail {
  public void setSender(String sender);
  public void setReceiver(String receiver);
  public void setContent(IContent content);
}
interface IContent {
  public String getAsString();
}
class Content implements IContent{
  ...
}
class Email implements IEmail {
  public void setSender(String sender) { ... }
  public void setReceiver(String receiver) { ... }
  public void setContent(IContent content) { ... }
}
```

- Si on change de protocole, on modifie la classe Email.
- Si on ajoute un type de contenu, on modifie la classe Content.

Open Close Principle (OCP)

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.

OCP : 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;
  }
}
```

tiré de http://www.oodesign.com/open-close-principle.html

OCP : 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
  }
}
```

The Liskov Substitution Principle (LSP)

Principe de substitution de Liskov

Une objet d'une classe dérivée peut toujours se substituer à un objet de la classe mère.

La classe des Carrés ne peut pas dériver de la classe des Rectangles

- Dans un rectangle, on peut modifier la largeur indépendamment de la longueur.
- Dans un carré , la modification de la largeur entraîne la modification de la longueur. La méthode suivante lève une exception pour un carré.

```
void invariant(Rectangle& r) {
  r.setHeight(200);
  r.setWidth(100);
  assert(r.getHeight() == 200 && r.getWidth() == 100);
}
```

tiré de http://stackoverflow.com/questions/56860/what-is-the-liskov-substitution-principle.

Interface Segregation Principle (ISP)

Séparation des interfaces

On ne doit pas implémenter des interfaces inutilisées.

```
// ISP bad exemple
public interface IMachine {
   public void print();
   public void staple();
   public void scan();
   public void photoCopy();
}
```

tiré de http://javatechig.com/design-patterns/interface-segregation-principle.

ISP : version améliorée

```
// ISP good example
public interface IPrinter {
  public void print();
}
public interface IScanner {
  public void fax();
}
public interface IStapler {
  public void staple();
}
public interface IPhotoCopier {
  public void photoCopy();
}
```

Dependency Inverse Principle (DIP)

Inversion des dépendances

Ce ne sont pas les modules de haut niveau qui dépendent des modules de bas niveau. On doit pouvoir réutiliser les modules de haut niveau si on change le bas niveau.

```
// Dependency Inversion Principle - Bad example
class Manager {
  Worker worker;
  public void setWorker(Worker w) {
  worker = w;
  }
  public void manage() {
    worker.work();
  }
}
class Worker {
  public void work() { // ....working }
}
class SuperWorker {
  public void work() { //.... working much more }
}
```

tiré de http://www.oodesign.com/dependency-inversion-principle.html

DIP : version améliorée

```
// Dependency Inversion Principle - Good example
class Manager {
  IWorker worker;
  public void setWorker(IWorker w) {
    worker = w;
  }
  public void manage() {
    worker.work();
  }
}
interface IWorker {
  public void work();
}
class Worker implements IWorker{
  public void work() { // ....working }
}
class SuperWorker  implements IWorker{
  public void work() { //....  working much more }
}
```

SOLID [3]

Les principes SOLID de R.C. Martin

- Single Responsibility Principle
- Open Close Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inverse Principle

3. voir `http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod`

Autres bonnes pratiques

- Program to Interface Not Implementation.
- Don't Repeat Yourself.
- Encapsulate What Varies.
- Depend on Abstractions, Not Concrete classes.
- Least Knowledge Principle.
- Favor Composition over Inheritance.
- Hollywood Principle (Don't call us. We'll call you.).
- Apply Design Pattern wherever possible.
- Strive for Loosely Coupled System.
- Keep it Simple and Sweet / Stupid.

tiré de `http://www.codeproject.com/Articles/567768/Object-Oriented-Design-Principles`

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 ⇒ Bug ⇒ Défaillance

Visibilité

Test en boite 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 boite 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 dévéloppement

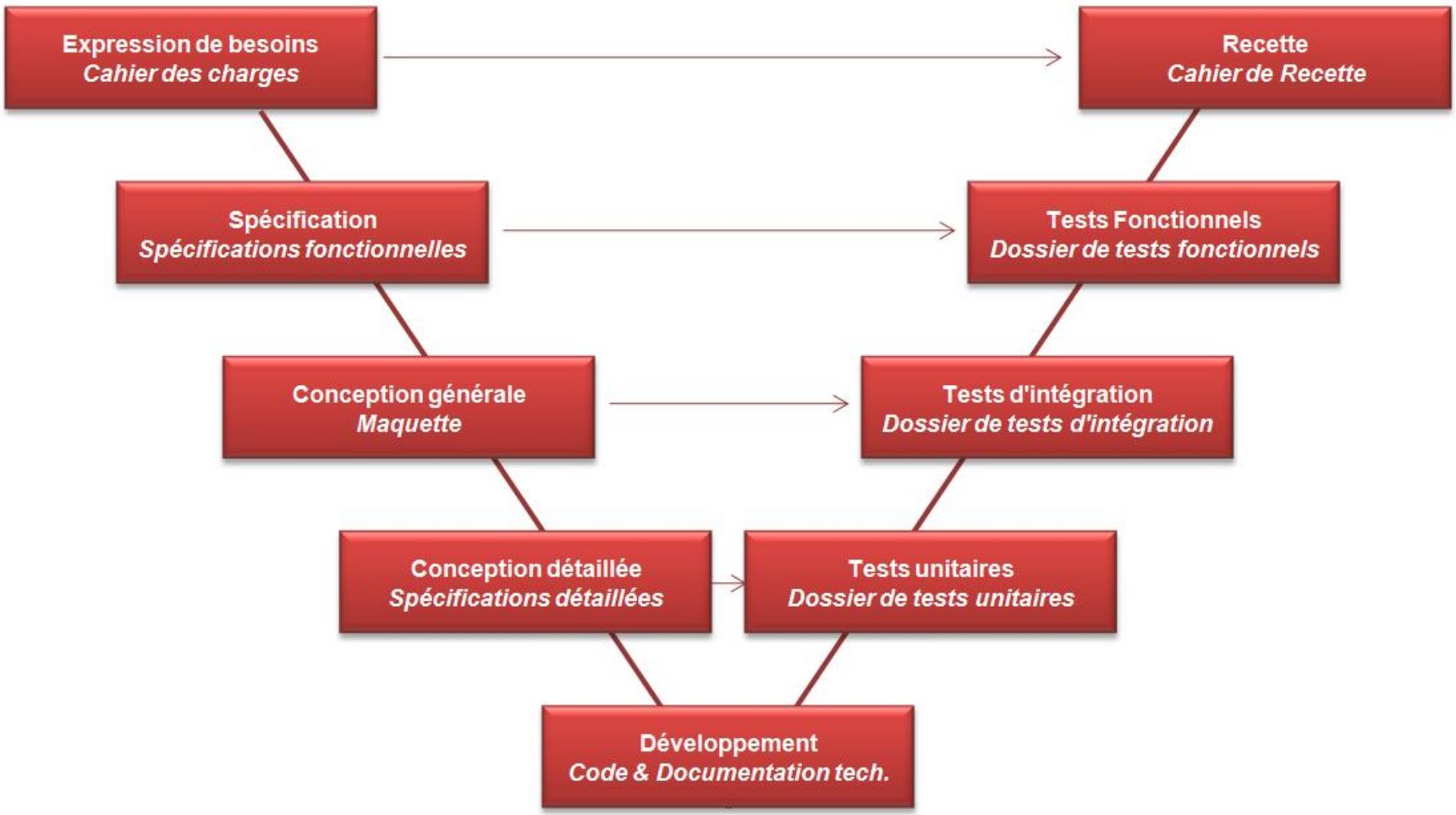


Image provenant de `http://www.aggil.com/`

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 boite 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 [4]

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

4. 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

fail(String message)	fait échouer le test
assertTrue(**boolean** condition)	vrai si condition vaut vrai
assertEquals(expected, actual)	vrai si les valeurs sont égales
assertEquals(expected, actual, tolerance)	vrai si les valeurs sont proches
assertNull(object)	vrai si **null**
assertNotNull(object)	vrai si différent de **null**
assertSame(expected, actual)	vrai si référencent le même objet
assertNotSame(expected, actual)	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.upec.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

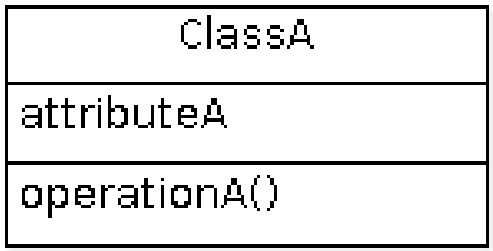
```
package fr.upec.test;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;
import org.junit.Before;
import org.junit.Test;

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 [5]

- 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 a la première erreur.

5. inspiré par `http://www.liafa.jussieu.fr/~sighirea/cours/methtest/c_JUnit.pdf`

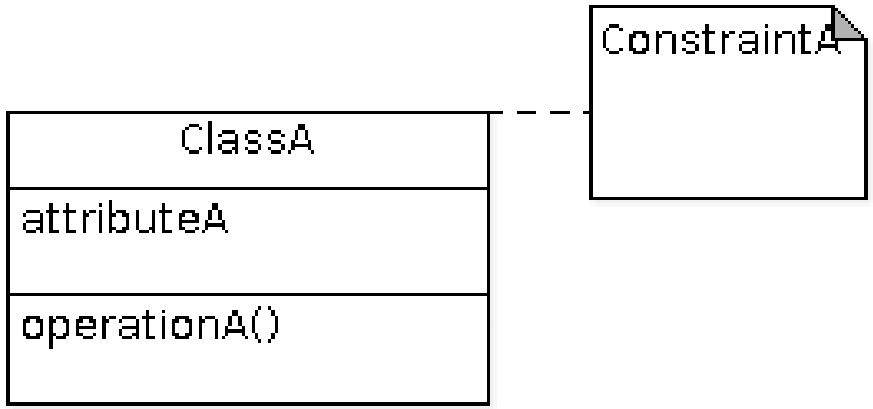
Représentation d'une classe



Visibilité des attributs et des opérations

- + : public
- − : private
- # : protected
- sinon : package

Note



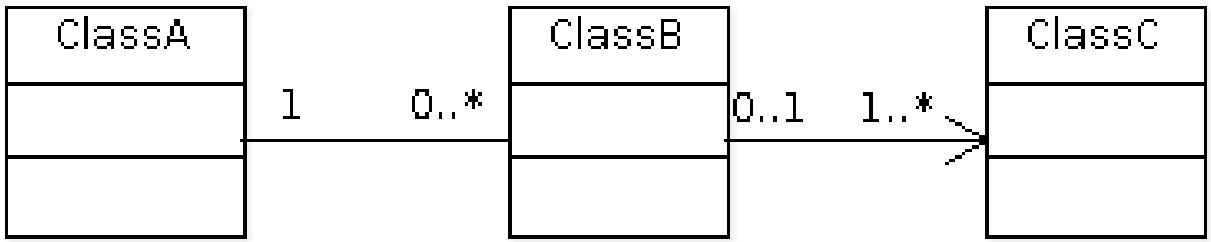
Note

Contrainte associée à une classe ou à une opération.

Association et navigabilité

Association

Relation binaire ou *n*-aire entre classes. On peut indiquer les multiplicités c'est à dire le nombre d'objets susceptibles d'apparaître dans la relation.

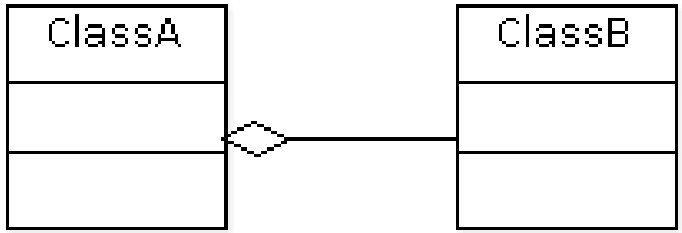


Navigabilité (flèche)

A partir d'un objet de la classe B, on peut accéder aux objets de la classe C. Mais pas l'inverse.

- Par défaut : navigabilité dans les deux sens.

Aggrégation

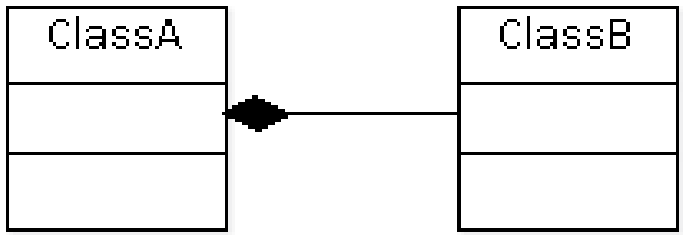


Aggrégation

Relation de type tout/partie : un objet de la classe B est un composant d'un objet de la classe A.

- La création (destruction) des objets de la classes A est indépendante de la création (destruction) des objets de la classe B.
- Un même composant peut appartenir à différents objets de type tout.

Composition

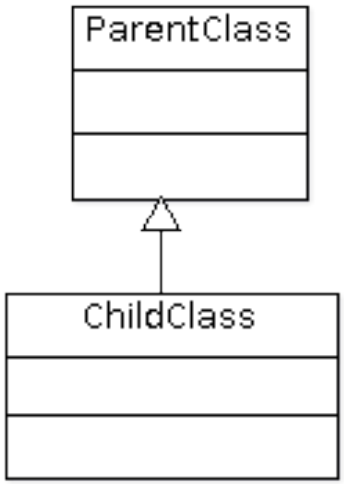


Composition

Aggrégation forte : un objet de la classe A contient les composants de la classe B.

- La création (destruction) des objets de la classes A entraîne la création (destruction) de ses composants.
- Un composant n'appartient qu'à un seul objet de de la classe A.

Héritage



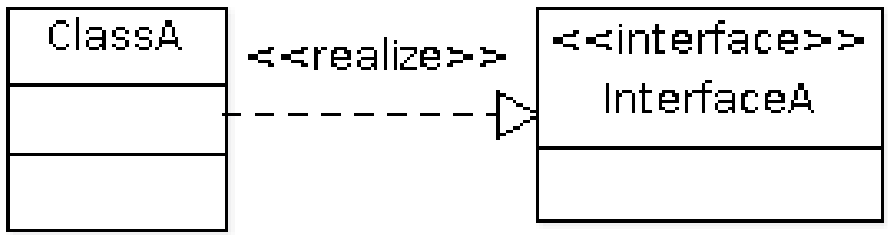
Spécialisation

Définir une classe enfant en ajoutant des champs et des opérations spécifiques.

Généralisation

Définir une classe parent pour factoriser des éléments communs aux classes enfants.

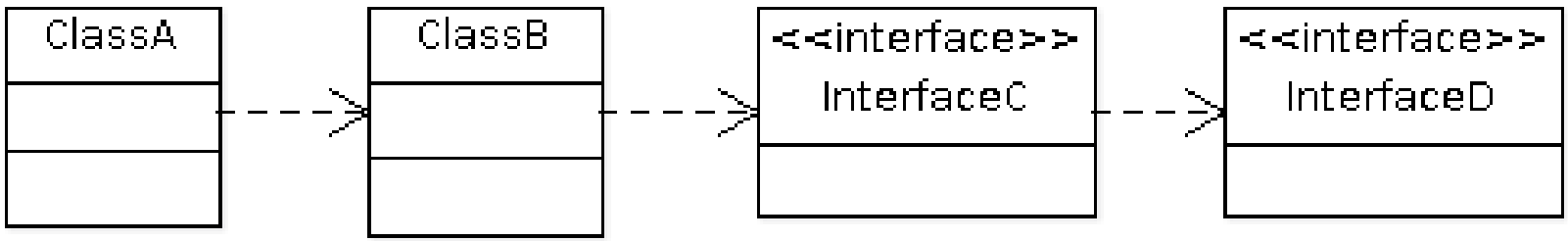
Réalisation



Interface

Définit les éléments qui permettent d'utiliser les objets des classes qui implémentent l'interface.

Dépendance



Dépendance

- Un changement dans ClassB peut avoir des répercussions dans ClassA.
- L'interface InterfaceC utilise InterfaceD.