Programmation, Génie Logiciel, Preuves

Langage Java

Zoubida Kedad, Stéphane Lopes

zoubida.kedad@uvsq.fr, stephane.lopes@uvsq.fr

2015–2016

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES
UFR DES SCIENCES
université PARIS-SACLAY

Table des matières

Chapitre 1

Préambule

Sommaire

1.1 Objectifs et prérequis

Objectifs du cours

— Maîtriser les bases de la conception orientée-objet
— Connaître différentes approches pour la persistance des objets
— Mettre en évidence les liens entre modèles (UML) et implémentation

1

Prérequis

— Connaître un langage de programmation objet
— Connaître la notation UML
— Connaître les outils de développement de base (GIT, MAVEN, ...)

2

1.2 Plan

Plan général

— Vue d'ensemble des concepts objets
— Principes de conception orientée-objet
— Persistance des objets
— Liens entre modèles et implémentation

3

1.3 Bibliographie

HUNT, Andrew et David THOMAS (2001). *The Pragmatic Programmer*. the Pragmatic Bookshelf. URL : http://www.pragprog.com/titles/tpp/the-pragmatic-programmer.

Chapitre 2

Vue d'ensemble des concepts objets

Sommaire

2.1 Objet

2.1.1 Système orienté objet

Système orienté objet

- Lors de son exécution, un *système OO* est **un ensemble d'objets qui interagissent**
- Les objets forment donc l'*aspect dynamique* (à l'exécution) d'un système OO
- Ces objets représentent soit
  - des entités du monde réel (⇒ ce sont donc des modèles d'entités réelles), soit
  - des objets « techniques » nécessaires durant l'exécution du programme.

4

2.1.2 Objet

Objet

- Un *objet* est formé de deux composants indissociables
  - son *état*, i.e. les valeurs prises par des variables le décrivant (*propriétés*)
  - son *comportement*, i.e. les opérations qui lui sont applicables
- Un objet est une *instance* d'une *classe*

— Un objet peut avoir plusieurs types, i.e. supporter plusieurs interfaces

Exemple
Des points et des cercles
(voir figure 2.1).

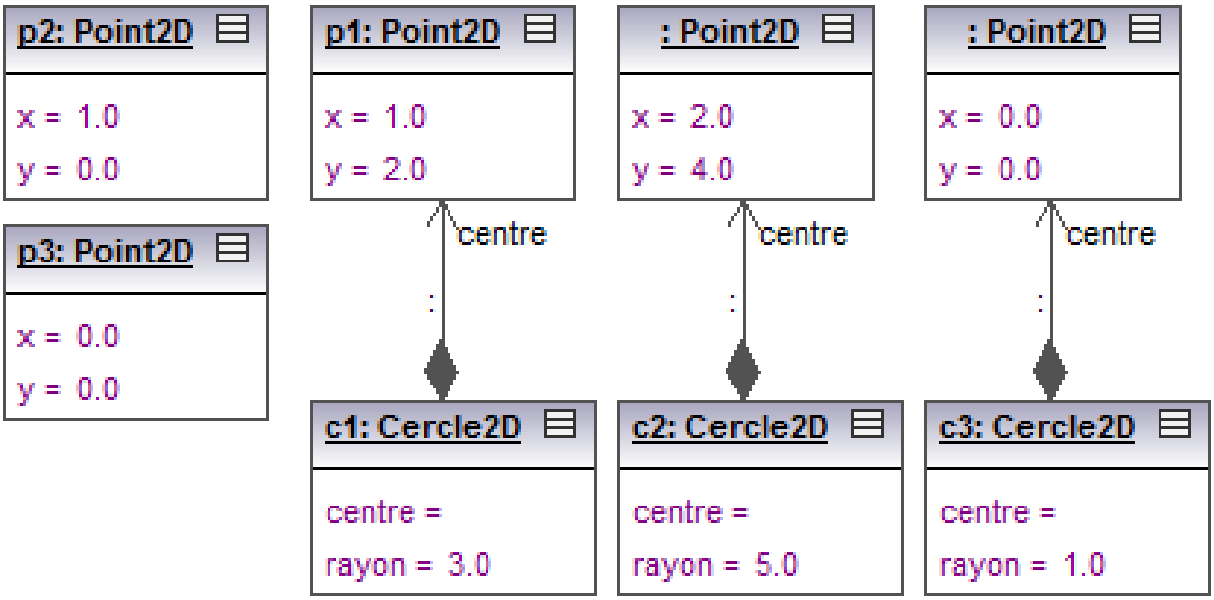

FIGURE 2.1 – Diagramme d'objets UML.

— Les objets `p1`, `p2` et `p3` sont des instances de la classe `Point2D` et deux autres objets `Point2D` sont anonymes
— Les objets `c1`, `c2` et `c3` sont des instances de la classe `Cercle2D`
— L'état de chaque objet est représenté par la valeur de ses propriétés
— L'attribut `centre` est une référence sur un objet `Point2D`
— Le comportement n'est pas représenté au niveau des objets
  — une opération est invoquée par rapport à un objet
  — mais elle est rattachée à la classe car le code est partagé par tous les objets d'une classe
— L'objet `p3` et le point anonyme de coordonnée (0, 0) sont égaux mais pas identiques

Exemple
Des points et des cercles en Java

```
Point2D p1 = new Point2D(1.0, 2.0);
Point2D p2 = new Point2D(1.0);
Point2D p3 = new Point2D();
Point2D unAutreP3 = p3;
  assert p3 == unAutreP3; // 2 points identiques

Cercle2D c1 = new Cercle2D(p1, 3.0);
Cercle2D c2 = new Cercle2D(new Point2D(2.0, 4.0), 5.0);
Cercle2D c3 = new Cercle2D();
```

Listing 2.1 – Instanciation de cercles et de points en Java

— Pour `p1, p2, p3` : déclaration de la variable, création de l'objet et initialisation de la variable en « une seule étape »
— Pour `unAutreP3` : déclaration de variable, pas de création d'objet, initialisation à partir de la référence sur `P3`

2.1.3 Message

Communication entre objets

— Un objet solitaire n'a que peu d'intérêt ⇒ différents objets doivent pouvoir interagir
— Un *message* est un moyen de communication (d'interaction) entre objets
— *Les messages sont les seuls moyens d'interaction entre objets* ⇒ l'état interne ne doit pas être manipulé directement
— Le (ou les) type(s) d'un objet détermine les messages qu'il peut recevoir

8

Message

— Un message est une requête envoyée à un objet pour demander l'exécution d'une opération
— Un message comporte trois composants :
  — *l'objet auquel il est envoyé* (le destinataire du message),
  — le nom de l'opération à invoquer,
  — les paramètres effectifs.

9

Exemple

Échange de messages

(voir figure 2.2).

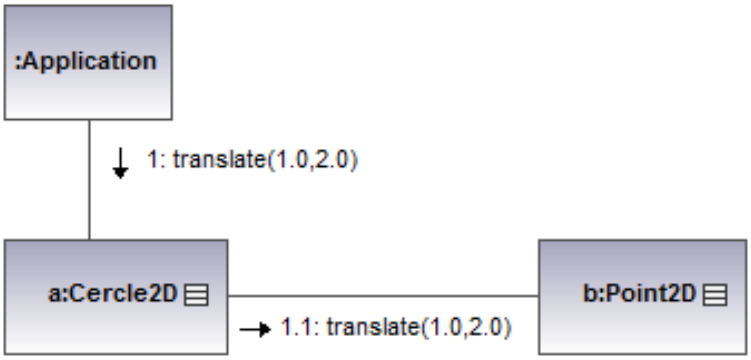

FIGURE 2.2 – Diagramme de communication UML.

— L'objet `Application` envoie un message à une instance `a` de `Cercle2D`
— Le message se traduit par l'exécution de l'opération `translate(1.0, 2.0)` par l'objet `a`
— Durant cette exécution, `a` envoie un message à l'objet `b` de classe `Point2D` (translater le cercle revient à translater son centre)
— La numérotation des messages reflète leur ordre et leur niveau d'imbrication

10

Exemple

Échange de messages en Java

— lors de l'exécution d'une opération de l'application, envoi de `a.translate(1.0, 2.0)`
  — lors de l'exécution de `translate` du cercle `a`, envoi de `b.translate(1.0, 2.0)`
    — exécution de `translate` du point `b`

11

2.2 Classe

2.2.1 Classe

Classe

— Une *classe* est un « modèle » (un « moule ») pour une catégorie d'objets structurellement identiques
  — Par exemple, la *voiture immatriculée 123456* est une instance de la classe *Voiture*
  — L'état et le comportement des voitures sont communs à toutes les voitures mais l'état courant de chaque voiture est indépendant des autres

  - Chaque instance aura ses propres copies des variables d'instances
- Une classe définit donc l'implémentation d'un objet (son état interne et le codage de ses opérations)
- L'ensemble des classes décrit l'*aspect statique* d'un système OO

12

Composition d'une classe

- Une classe comporte :
  - la définition des *attributs* (ou *variables d'instance*),
  - la *signature* des opérations (ou *méthodes*),
  - la *réalisation* (ou *définition*) des méthodes.
- Chaque instance aura sa propre copie des attributs
- La signature d'une opération englobe son nom et le type de ses paramètres
- L'ensemble des signatures de méthodes représente l'interface de la classe (publique)
- L'ensemble des définitions d'attributs et de méthodes forme l'implémentation de la classe (privé)

13

Exemple

Les classes `Cercle2D` *et* `Point2D` *(mode abrégé)*

(voir figure 2.3).

FIGURE 2.3 – Diagramme de classes UML (mode abrégé).

14

Exemple

Les classes `Cercle2D` *et* `Point2D`

(voir figure 2.4).

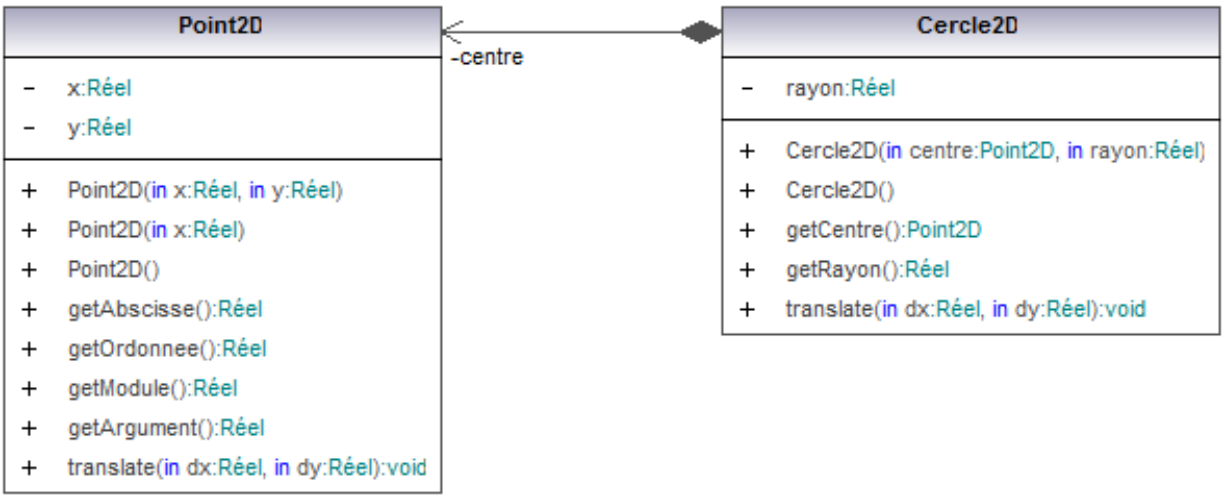

FIGURE 2.4 – Diagramme de classes UML.

- Un rectangle représente une classe
  - 1erpavé : nom de la classe
  - 2^{e}pavé : attributs
  - 3^{e}pavé : signature des méthodes
- Accès aux membres

  - + = public, # = protégé, – = privé, ~ = paquetage
  - en général, les attributs sont privés et les méthodes publiques
- `Point2D` et `Cercle2D` sont des constructeurs (surcharge), `getXXX` sont des accesseurs, `translate` est un mutateur.
- Exemple d'invariant : « le rayon du cercle doit toujours être positif »

Exemple

La classes `Cercle2D` en Java

```
class Cercle2D implements Deplacable {
    /** Le centre du cercle. */
    private Point2D centre;

    /** Le rayon du cercle */
    private double rayon;

    /**
     * Initialise un cercle avec un centre et un rayon.
     * @param centre Le centre.
     * @param rayon Le rayon.
     */
    public Cercle2D(Point2D centre, double rayon) { /* ... */ }

    /**
     * Initialise un cercle centre a l'origine et de rayon 1.
     */
    public Cercle2D() { /* ... */ }

    public Point2D getCentre() { /* ... */ }
    public double getRayon() { /* ... */ }

    /**
     * Translate le cercle.
     * @param dx deplacement en abscisse.
     * @param dy deplacement en ordonnees.
     */
    public void translate(double dx, double dy) { /* ... */ }
}
```

Listing 2.2 – La classes `Cercle2D` en Java

2.2.2 Classe et objet

Instanciation d'une classe

- Le mécanisme d'*instanciation* permet de créer des objets à partir d'une classe
- Chaque objet est une instance d'une classe
- Lors de l'instanciation,
  - de la mémoire est allouée pour l'objet,
  - l'objet est initialisé afin de respecter l'invariant de la classe.

Exemple

Classe et objet

(voir figure 2.5).

- Les objets `p1`, `p2` et `p3` sont des instances de la classe `Point2D`
- Ce type de lien est représenté par le *stéréotype « instanceof »* en UML
- On représente rarement classes et objets sur le même schéma (pas le même point de vue)
- *Attention à bien différencier classe et objet*

2.2.3 Classe et type

Type

- Un *type* est un modèle abstrait réunissant à un haut degré les traits essentiels de tous les êtres ou de tous les objets de même nature
- En informatique, un *type (de donnée)* spécifie :
  - l'ensemble des valeurs possibles pour cette donnée (définition en *extension*),
  - l'ensemble des opérations applicables à cette donnée (définition en *intention*).
- Un type spécifie l'*interface* par laquelle une donnée peut être manipulée

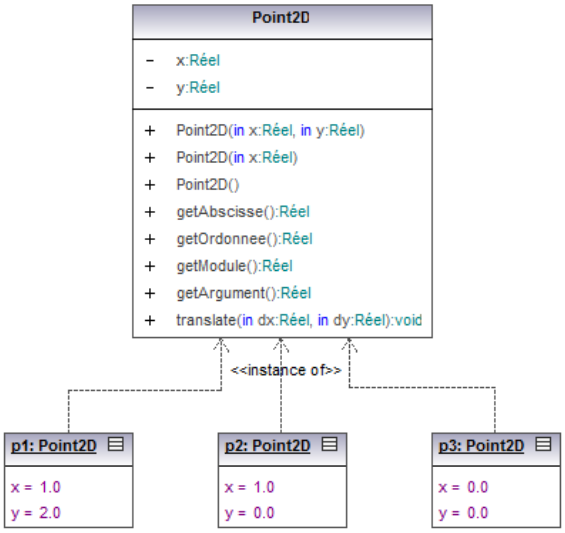

FIGURE 2.5 – Lien entre classe et objet.

Exemple
Représentation d'un type comme une interface
(voir figure 2.6).

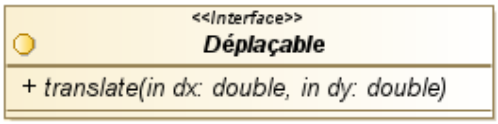

FIGURE 2.6 – Représentation d'un type comme une interface.

Exemple
L'interface `Deplacable` *en Java*

```
interface Deplacable {
    /**
     * Translate l'objet.
     * @param dx deplacement en abscisse.
     * @param dy deplacement en ordonnees.
     */
    void translate(double dx, double dy);
}
```

Listing 2.3 – L'interface `Deplacable` en Java

Classe et type
- Une classe implémente un ou plusieurs types, i.e. respecte une ou plusieurs interfaces
- Un objet peut avoir plusieurs types mais est une instance d'une seule classe
- Des objets de classes différentes peuvent avoir le même type

Exemple
Interface et classe
(voir figure 2.7).

```
class Cercle2D implements Deplacable {
```

Listing 2.4 – `Cercle2D` implémente `Deplacable` en Java

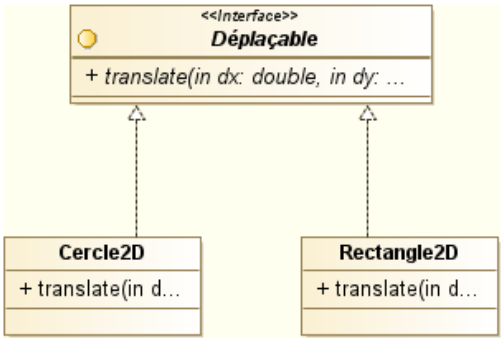

FIGURE 2.7 – Relation entre classe et interface.

Exercice
Modélisation d'objets et de classes

On veut modéliser des robots se déplaçant sur un terrain. Ce terrain est découpé en cases carrées repérées par deux coordonnées. Chaque case peut être vide ou contenir un mur ou un robot. Les robots sont très rudimentaires et ne disposent que d'une boussole. Ils ne connaissent donc que leur orientation (Nord, Est, Sud, Ouest). Un robot doit pouvoir avancer d'une case et tourner d'un quart de tour à droite. Un robot ne peut se déplacer d'une case à une autre que si la case de destination est vide.

1. Représenter avec la notation vue précédemment la classe `Robot`
2. Faire de même avec la classe `Terrain`
3. Représenter sur un diagramme de communication les échanges de messages pour le déplacement d'un robot
4. On suppose maintenant qu'un robot peut en détecter un autre qui passe devant lui. Par exemple, quand un robot se déplace, il peut passer dans le champ de vision d'un autre. Ce dernier devra alors être averti. Modifier le diagramme de communication précédent pour y intégrer la détection des déplacements

25

2.3 Héritage

2.3.1 Héritage

Héritage

— L'*héritage* permet de définir l'implémentation d'une classe à partir de l'implémentation d'une autre
— Ce mécanisme permet, lors de la définition d'une nouvelle classe, de ne préciser que ce qui change par rapport à une classe existante
— Une *hiérarchie de classes* permet de gérer la complexité, en ordonnant les classes au sein d'arborescences d'abstraction croissante
— Si Y hérite de X, on dit que Y est une classe *fille* (*sous-classe*, *classe dérivée*) et que X est une classe *mère* (*super-classe*, *classe de base*)

30

Exemple
Rectangle et rectangle plein

(voir figure 2.8).

31

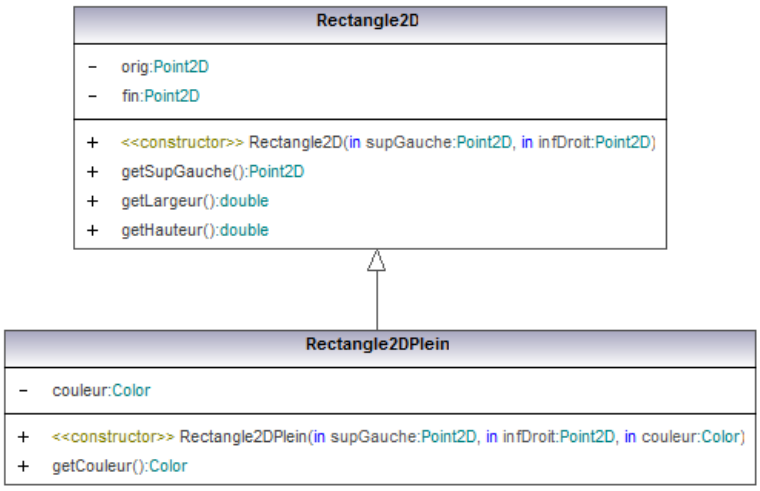

FIGURE 2.8 – Héritage entre rectangle plein et rectangle.

Exercice

L'héritage

Deux nouveaux types de robot sont créés :

— les transporteurs peuvent ramasser un objet, le transporter et le déposer,

— les destructeurs peuvent détruire ce qui se trouve sur la case devant eux.

1. Modéliser les robots
2. Soit le terrain suivant

T S	M	
D E	M	
	M	

(M = mur, T = transporteur, D = destructeur, S = sud, E = est). Écrire les instructions permettant de créer ce terrain puis de déplacer l'objet se trouvant en (0, 0) en (2, 2).

3. Donner l'implémentation des deux classes `Transporteur` et `Destructeur`.

32

Exercice (solutions)

Modélisation

(voir figure 2.9).

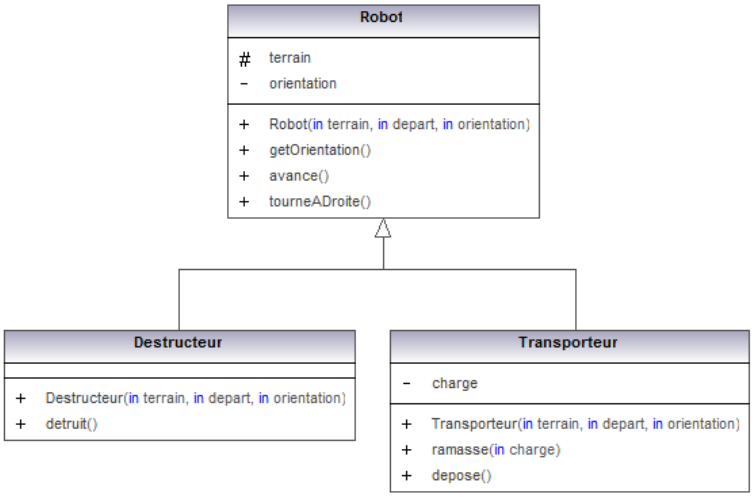

FIGURE 2.9 – Transporteur et destructeur.

33

Exercice (solutions)

Déplacement des robots

```
        // Création du terrain
        Position[] murs = { new Position(1, 0), new Position(1, 1),
                            new Position(1, 2) };
        Terrain terrain = new Terrain(3, 3, murs);

        // Création des robots
        Transporteur transp = new Transporteur(terrain, new Position(0, 0),
                                               Direction.SUD);
        Destructeur destr = new Destructeur(terrain, new Position(0, 1),
                                            Direction.EST);

        // Actions des robots
        destr.detruit();
        destr.tourneADroite();
        destr.avance();
        transp.ramasse(new Object());
        transp.avance();
        transp.tourneADroite(); transp.tourneADroite(); transp.tourneADroite();
        transp.avance();
        transp.avance();
        transp.tourneADroite();
        transp.avance();
        transp.depose();
```

Listing 2.5 – Déplacement des robots

Exercice (solutions)

Classe `Transporteur` 1/2

```
/**
 * Cette classe représente un type de robot particulier.
 * Ces robots ont la capacité de transporter un objet.
 *
 * @version oct. 2008
 * @author Stéphane Lopes
 */
class Transporteur extends Robot {
    /** L'objet transporté. */
    private Object charge;

    /**
     * Initialise le robot.
     * @param terrain terrain sur lequel le robot va évoluer.
     * @param depart position de départ du robot.
     * @param direction orientation du robot.
     */
    public Transporteur(Terrain terrain, Position depart,
                        Direction orientation) {
        super(terrain, depart, orientation);
    }
```

Listing 2.6 – Classe `Transporteur` (part. 1)

Exercice (solutions)

Classe `Transporteur` 2/2

```
    /**
     * Prends un objet sur le terrain.
     * @param charge objet à transporter.
     */
    public void ramasse(Object charge) {
        this.charge = charge;
    }

    /**
     * Dépose l'objet transporté.
     * @return l'objet transporté.
     */
    public Object depose() {
        Object tmp = charge;
        charge = null;
        return tmp;
    }
}
```

Listing 2.7 – Classe `Transporteur` (part. 2)

Exercice (solutions)

Classe `Destructeur`

```
/**
 * Cette classe représente un type de robot particulier.
 * Ces robots ont la capacité de détruire un obstacle
 * se trouvant devant eux.
 *
 * @version oct. 2008
```

```
 * @author Stéphane Lopes
 */
class Destructeur extends Robot {
    /**
     * Initialise le robot.
     * @param terrain terrain sur lequel le robot va évoluer.
     * @param depart position de départ du robot.
     * @param direction orientation du robot.
     */
    public Destructeur(Terrain terrain, Position depart,
                       Direction orientation) {
        super(terrain, depart, orientation);
    }

    /**
     * Supprime le contenu de la case se trouvant devant le robot
     */
    public void detruit() {
        terrain.detruitCaseDevant(this);
    }
}
```

Listing 2.8 – Classe `Destructeur`

— L'attribut `terrain` doit être déclaré `protected` dans `Robot` pour être accessible au `Destructeur`. 37

2.3.2 Polymorphisme

Polymorphisme

— Le *polymorphisme* est l'aptitude qu'ont des objets à réagir différemment à un même message
— L'intérêt est de pouvoir gérer une collection d'objets de façon homogène tout en conservant le comportement propre à chaque objet
— Une méthode commune à une hiérarchie de classe peut prendre plusieurs formes dans différentes classes
— Une sous-classe peut *redéfinir* une méthode de sa super-classe pour spécialiser son comportement
— Le choix de la méthode à appeler est retardé jusqu'à l'exécution du programme (*liaison dynamique ou retardée*) 38

Exemple

Une description pour le rectangle et le rectangle plein
(voir figure 2.10).

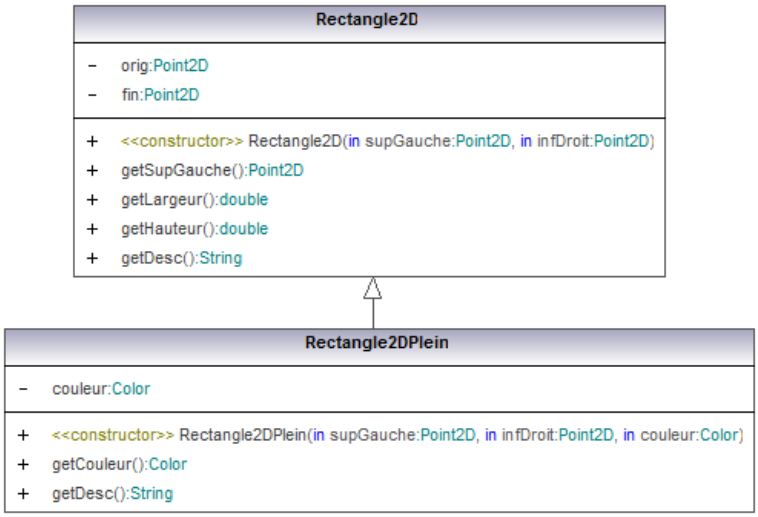

FIGURE 2.10 – Polymorphisme entre rectangle plein et rectangle.

39

Exercices

Le polymorphisme et la redéfinition

Une amélioration importante a été apportée aux robots : ils sont maintenant programmables. Chaque type de robot possède son propre comportement. Quand ils en reçoivent l'ordre, ils exécutent l'action programmée (un ensemble d'instructions élémentaires). Plusieurs robots de types différents se trouvent sur le terrain. On veut pouvoir déclencher l'exécution du programme de l'ensemble des robots.

1. Modéliser cet énoncé
2. Implémenter les changements dans les classes `Robot`, `Transporteur` et `Destructeur`
3. Écrire un programme déclenchant l'exécution de l'action pour l'ensemble des robots

40

Exercice (solutions)
Modélisation
(voir figure 2.11).

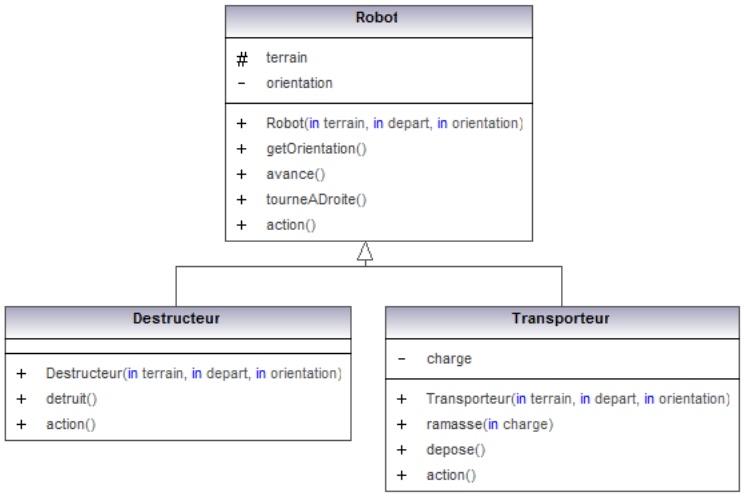

FIGURE 2.11 – Modélisation des robots programmables.

41

Exercice (solutions)
La classe `Robot`

```
public class Robot {
    /** Terrain sur lequel évolue le robot. */
    protected Terrain terrain;

    /** Orientation du robot. */
    private Direction orientation;

    /**
     * Initialise le robot.
     * @param terrain terrain sur lequel le robot va évoluer
     * @param direction orientation du robot
     */
    public Robot(Terrain terrain, Position depart, Direction orientation) {
        assert terrain != null;
        this.terrain = terrain;
        terrain.ajouterRobot(this, depart);
        this.orientation = orientation;
    }

    public Direction getOrientation() { return orientation; }
    public boolean avance() { return terrain.avancerRobot(this); }
    public void tourneADroite() { orientation = orientation.next(); }

    public void action() {
        for (int i = 0; i < 4; ++i) {
            avance(); avance();
            tourneADroite();
        }
    }
}
```

Listing 2.9 – La classe `Robot`

42

Exercice (solutions)

La classe ***Transporteur***

```
class Transporteur extends Robot {
    /** L'objet transporté. */
    private Object charge;

    public Transporteur(Terrain terrain, Position depart,
      Direction orientation) {
        super(terrain, depart, orientation);
    }

    public void ramasse(Object charge) { this.charge = charge; }

    public Object depose() {
        Object tmp = charge;
        charge = null;
        return tmp;
    }

    public void action() {
        super.action();
        ramasse(new Object());
        avance();
        depose();
    }
}
```

Listing 2.10 – La classe `Transporteur`

Exercice (solutions)

La classe ***Destructeur*** *1/2*

```
/**
 * Cette classe représente un type de robot particulier.
 * Ces robots ont la capacité de détruire un obstacle
 * se trouvant devant eux.
 *
 * @version oct. 2008
 * @author Stéphane Lopes
 */
class Destructeur extends Robot {
    /**
     * Initialise le robot.
     * @param terrain terrain sur lequel le robot va évoluer.
     * @param depart position de départ du robot.
     * @param direction orientation du robot.
     */
    public Destructeur(Terrain terrain, Position depart,
            Direction orientation) {
        super(terrain, depart, orientation);
    }
```

Listing 2.11 – La classe `Destructeur` (part. 1)

Exercice (solutions)

La classe ***Destructeur*** *2/2*

```
    /**
     * Supprime le contenu de la case se trouvant devant le robot
     */
    public void detruit() {
        terrain.detruitCaseDevant(this);
    }

    /*
     * Décrit la programmation du robot.
     */
    public void action() {
        for (int i = 0; i < 4; ++i) {
            detruit(); avance();
        }
    }
}
```

Listing 2.12 – La classe `Destructeur` (part. 2)

Exercice (solutions)

La classe ***Application***

```
        // Création du terrain
        Terrain terrain = new Terrain(4, 5, null);

        // Création des robots
        final int NB_ROBOTS = 3;
        Robot[] robots = new Robot[NB_ROBOTS];
        robots[0] = new Robot(terrain, new Position(0, 0),
                              Direction.EST);
```

```
        robots[1] = new Transporteur(terrain, new Position(1, 1),
                                     Direction.EST);
        robots[2] = new Destructeur(terrain, new Position(3, 0),
                                    Direction.SUD);

// Exécution des actions des robots
        for (int i = 0; i < NB_ROBOTS; ++i) {
            robots[i].action();
        }
```

Listing 2.13 – La classe `Application`

46

2.3.3 Classe abstraite

Classe abstraite

- Une *classe abstraite* représente un concept abstrait qui ne peux pas être instancié
- En général, son comportement ne peut pas être intégralement implémenté à cause de son niveau de généralisation
- Elle sera donc seulement utilisée comme classe de base dans une hiérarchie d'héritage

47

Exemple

La hiérarchie d'héritage des figures

(voir figure 2.12).

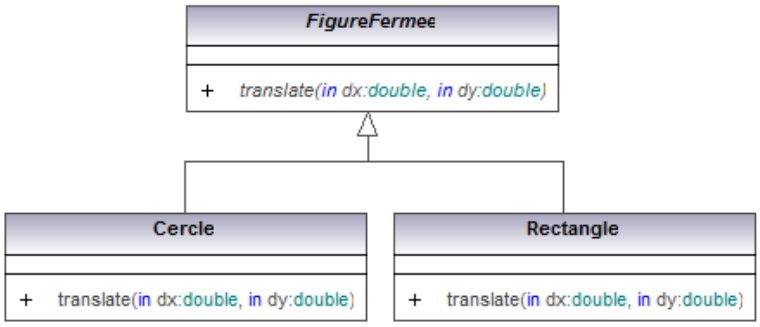

FIGURE 2.12 – Hiérarchie d'héritage des figures.

- Deux façons de représenter une classe abstraite : `abstract` entre `{}` ou nom de la classe en italique
- Méthodes abstraites en italique

48

Exercices

Les classes abstraites

On souhaite maintenant ajouter sur le terrain un type de case sensible à la charge (pont par exemple). Chaque élément mobile (robot ou objet transportable) devra donc posséder un poids. Le poids d'une instance de robot sera toujours de 1 unité. Une instance de destructeur aura un poids de 2 unités de plus que le robot. Une instance de transporteur aura un poids de 1 unités de plus que le robot auquel il faudra ajouter le poids de l'objet transporté. Le poids par défaut des objets transportables est de 1 unité mais chaque objet pourra avoir un poids différent précisé lors de sa création. On souhaite pouvoir connaître le poids de tout élément se trouvant sur le terrain.

1. Modéliser la prise en compte du poids au niveau des éléments mobiles
2. Implémenter les classes correspondantes

49

Exercice (solutions)

Modélisation

(voir figure 2.13).

50

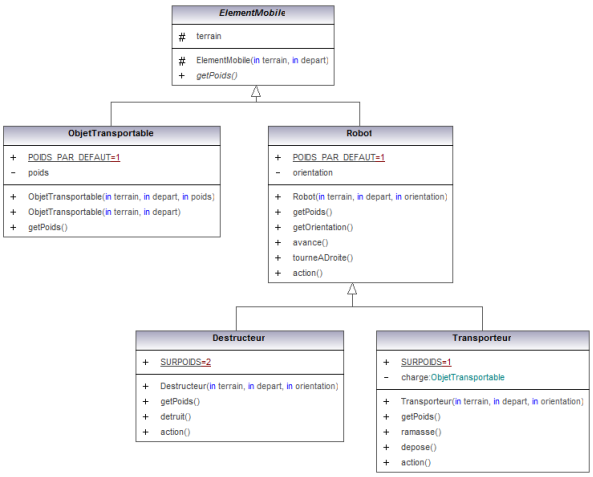

FIGURE 2.13 – Modélisation du poids.

Exercice (solutions)

La classe ***ElementMobile***

```java
public abstract class ElementMobile {
    /** Terrain sur lequel se trouve l'élément. */
    protected Terrain terrain;

    /**
     * Place l'élément sur un terrain à une position donnée.
     * @param terrain terrain sur lequel l'élément va évoluer
     * @param depart position de départ sur le terrain
     */
    protected ElementMobile(Terrain terrain, Position depart) {
        assert terrain != null && depart != null;
        this.terrain = terrain;
        terrain.ajouter(this, depart);
    }

    /**
     * Retourne le poids de l'élément.
     * @return le poids de l'élément.
     */
    public abstract int getPoids();
}
```

Listing 2.14 – La classe `ElementMobile`

51

Exercice (solutions)

La classe ***ObjetTransportable*** *1/2*

```java
public class ObjetTransportable extends ElementMobile {
    /** Le poids par défaut des objets */
    public static final int POIDS_PAR_DEFAUT = 1;

    /** Le poids de l'objet */
    private int poids;

    /**
     * Initialise l'objet.
     * @param terrain terrain sur lequel l'objet se trouve
     * @param depart position de départ sur le terrain
     * @param poids le poids de l'objet.
     */
    public ObjetTransportable(Terrain terrain, Position depart, int poids) {
        super(terrain, depart);
        this.poids = poids;
    }

    /**
     * Initialise un objet de poids 1.
     * @param terrain terrain sur lequel l'objet se trouve
     * @param depart position de départ sur le terrain
     */
    public ObjetTransportable(Terrain terrain, Position depart) {
        this(terrain, depart, POIDS_PAR_DEFAUT);
    }
```

Listing 2.15 – La classe `ObjetTransportable`

52

Exercice (solutions)

La classe ***ObjetTransportable*** *2/2*

```
    /**
     * Retourne le poids de l'élément.
     * @return le poids de l'élément.
     */
    @Override
    public int getPoids() {
        return poids;
    }
}
```

Listing 2.16 – La classe `ObjetTransportable`

53

Exercice (solutions)

La classe `Robot` 1/2

```
/**
 * Cette classe représente un robot pouvant évoluer sur un terrain.
 *
 * @version oct. 2008
 * @author Stéphane Lopes
 */
public class Robot extends ElementMobile {
    /** Le poids par défaut des robots */
    public static final int POIDS_PAR_DEFAUT = 1;

    /** Orientation du robot. */
    private Direction orientation;

    /**
     * Initialise le robot.
     * @param terrain terrain sur lequel le robot va évoluer
     * @param depart position de départ sur le terrain
     * @param direction orientation du robot
     */
    public Robot(Terrain terrain, Position depart, Direction orientation) {
        super(terrain, depart);
        this.orientation = orientation;
    }
```

Listing 2.17 – La classe `Robot` (part. 1)

54

Exercice (solutions)

La classe `Robot` 2/2

```
    /**
     * Retourne le poids du robot.
     * @return le poids de l'élément.
     */
    @Override
    public int getPoids() {
        return POIDS_PAR_DEFAUT;
    }
```

Listing 2.18 – La classe `Robot` (part. 2)

55

Exercice (solutions)

La classe `Destructeur`

```
class Destructeur extends Robot {
    /** Surpoids des robots destructeurs */
    public static final int SURPOIDS = 2;

    /**
     * Initialise le robot.
     * @param terrain terrain sur lequel le robot va évoluer.
     * @param depart position de départ du robot.
     * @param direction orientation du robot.
     */
    public Destructeur(Terrain terrain, Position depart, Direction orientation) {
        super(terrain, depart, orientation);
    }

    /**
     * Retourne le poids du robot.
     * @return le poids de l'élément.
     */
    @Override
    public int getPoids() {
        return super.getPoids() + SURPOIDS;
    }
```

Listing 2.19 – La classe `Destructeur`

56

Exercice (solutions)
La classe `Transporteur` 1/2

```
/**
 * Cette classe représente un type de robot particulier.
 * Ces robots ont la capacité de transporter un objet.
 *
 * @version oct. 2008
 * @author Stéphane Lopes
 */
class Transporteur extends Robot {
    /** Surpoids des robots transporteurs */
    public static final int SURPOIDS = 1;

    /** L'objet transporté. */
    private ObjetTransportable charge;

    /**
     * Initialise le robot.
     * @param terrain terrain sur lequel le robot va évoluer.
     * @param depart position de départ du robot.
     * @param direction orientation du robot.
     */
    public Transporteur(Terrain terrain, Position depart, Direction orientation) {
        super(terrain, depart, orientation);
    }
```

Listing 2.20 – La classe `Transporteur` (part. 1)

57

Exercice (solutions)
La classe `Transporteur` 2/2

```
    /**
     * Retourne le poids du transporteur
     * éventuellement chargé.
     * @return le poids de l'élément.
     */
    @Override
    public int getPoids() {
        return super.getPoids() +
            SURPOIDS +
            (charge != null ? charge.getPoids() : 0);
    }
```

Listing 2.21 – La classe `Transporteur` (part. 2)

58

Exercice (solutions)
L'application

```
  // Création du terrain
  Terrain terrain = new Terrain(4, 5, null);

  // Création des éléments mobiles
  final int NB_ELEMENTS = 5;
  ElementMobile[] elements = new ElementMobile[NB_ELEMENTS];
  elements[0] = new Robot(terrain, new Position(0, 0), Direction.EST);
  elements[1] = new Transporteur(terrain, new Position(1, 1),
      Direction.EST);
  elements[2] = new Destructeur(terrain, new Position(3, 0),
      Direction.SUD);
  elements[3] = new ObjetTransportable(terrain,
      new Position(3, 4), 5);
  elements[4] = new ObjetTransportable(terrain,
      new Position(1, 2), 2);

  // Calcul du poids total
  int poidsTotal = 0;
  for (int i = 0; i < NB_ELEMENTS; ++i) {
    poidsTotal += elements[i].getPoids();
  }
  assert poidsTotal == (1+3+2+5+2): poidsTotal;
```

Listing 2.22 – L'application

59

2.3.4 Héritage multiple et à répétition

Héritage multiple et à répétition

- Un *héritage multiple* se produit lorsqu'une classe possède plusieurs super-classes
- Un *héritage à répétition* se produit lorsqu'une classe hérite plusieurs fois d'une même super-classe
- Ces types d'héritage peuvent provoquer des conflits aux niveaux des attributs et méthodes
  - deux classes de base peuvent posséder la même méthode,

— un attribut peut être hérité selon plusieurs chemins dans le graphe d'héritage.
— L'héritage multiple de classe n'est pas supporté par Java

Exemple
Héritage multiple et à répétition
(voir figure 2.14).

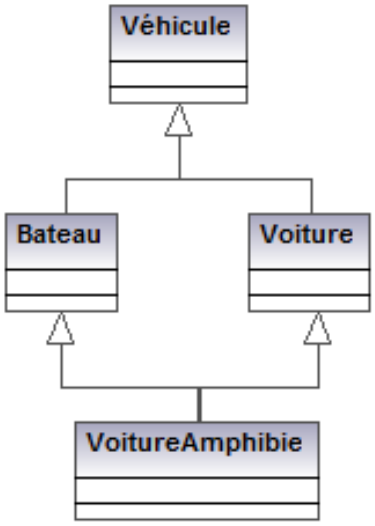

FIGURE 2.14 – Héritage multiple et à répétition.

2.3.5 Héritage et sous-typage

Sous-type

Sous-type
Un type T_1 est un *sous-type* d'un type T_2 si l'interface de T_1 contient l'interface de T_2

— Un sous-type possède une interface plus riche, i.e. au moins toutes les opérations du super-type
— De manière équivalente, l'extension du super-type contient l'extension du sous-type, i.e. tout objet du sous-type est aussi instance du super-type

Principe de substitution de Liskov

Principe de substitution de Liskov
Si pour chaque objet o_1 de type S, il existe un objet o_2 de type T tel que, pour tout programme P défini en terme de T, le comportement de P demeure inchangé lorsque o_1 est remplacé par o_2, alors S est un sous-type de T.

— Un objet du sous-type peut remplacer un objet du super-type sans que le comportement du programme ne soit modifié

Héritage et sous-typage
— L'héritage (ou *héritage d'implémentation*) est un mécanisme technique de réutilisation
— Le sous-typage (ou *héritage d'interface*) décrit comment un objet peut être utilisé à la place d'un autre
— Si Y est une sous-type de X, cela signifie que « Y est une sorte de X » (relation *IS-A*)
— Dans un langage de programmation, les deux visions peuvent être représentées de la même façon : le mécanisme d'héritage permet d'implémenter l'un ou l'autre
— En Java, les interfaces et l'héritage entre interface implémentent plus spécifiquement le concept de sous-typage

Exemple
Héritage d'implémentation et d'interface
(voir figure 2.15).

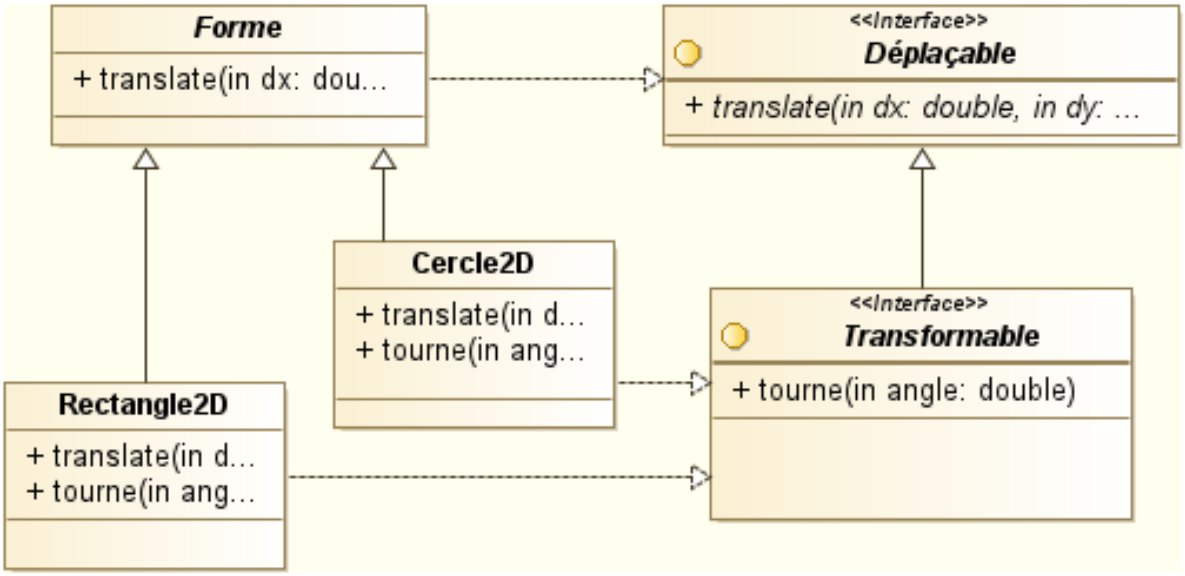

FIGURE 2.15 – Héritage d'implémentation et d'interface.

65

2.4 Module

Module
- Un *module* (ou *package)* est l'unité de base de décomposition d'un système
- Il permet d'organiser logiquement des modèles
- Un module s'appuie sur la notion d'*encapsulation*
  - publie une interface, i.e. ce qui est accessible de l'extérieur
  - utilise le principe de *masquage de l'information*, i.e. ce qui ne fait pas parti de l'interface est dissimulé

66

Utilité d'un module
- Sert de brique de base pour la construction d'une architecture
- Représente le bon niveau de granularité pour la réutilisation
- Est un *espace de noms* qui permet de gérer les conflits

67

Qualité d'un module
- La conception d'un module devrait conduire à un *couplage faible* et une *forte cohésion*

  couplage désigne l'importance des liaisons entre les éléments ⇒ *doit être réduit*

  cohésion mesure le recouvrement entre un élément de conception et la tâche logique à accomplir ⇒ *doit être élevé*, i.e. chaque élément est responsable d'une tâche précise

68

Exemple
Un diagramme de packages UML
(voir figure 2.16).
- Liens d'utilisation entre module
- D'autres types de liens sont également possible comme l'héritage

69

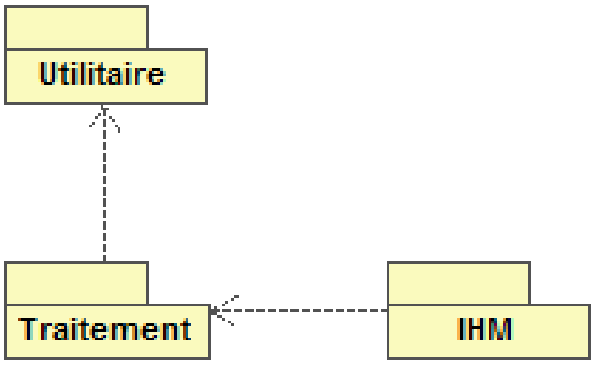

FIGURE 2.16 – Diagramme de packages UML.

Module en Java

— Le mot clé `package` placé en début de fichier permet l'ajout d'éléments dans un module
— Le mot clé `import` permet l'accès aux éléments d'un module

70

2.5 Exercices

Exercice 2.1 (Le premier exercice)
Dans cette exercice, . . .

Chapitre 3

Principes de conception orientée-objet

Sommaire

3.1 Introduction

Conception orientée-objet

— Lors de son exécution, un *système OO* est **un ensemble d'objets qui interagissent**
— La *conception orientée-objet* (COO) consiste donc à créer un *modèle* qui respecte les concepts objet

Difficultés de la conception orientée-objet

— Les concepts objets sont nombreux et complexes (attribut, méthode, objet, classe, héritage, ...)
⇒ plusieurs solutions sont en général envisageables
— Identifier la bonne solution est difficile
— Plusieurs symptômes voire métriques de qualité peuvent guider les choix

Programmer en Java ou en C# n'est pas concevoir objet !

— Seule une analyse objet conduit à une solution objet, i.e. qui respecte les concepts objet
— Le langage de programmation est un moyen d'implémentation qui ne garantit pas le respect des concepts objet

72

Symptôme d'une conception défectueuse

Rigidité résistance aux changements
— un simple changement provoque une cascade de modification dans les modules dépendants
— génère des réticences à se lancer dans des modifications

Fragilité tendance du logiciel à avoir des défaillances lors de changements
— défaillance même dans des régions non directement liées au changement
— rend la maintenance difficile

Immobiblité impossibilité de réutiliser des modules
— les modules sont tellement dépendants qu'il est très difficile de les utiliser dans un contexte différent

Viscosité un changement qui respecte la conception est plus difficile à réaliser qu'un bricolage (*hack*)

73

Anticiper les changements

— La cause des problèmes de conception est liée aux changements dans les besoins du client
— Les évolutions sont faites sans forcément respecter la conception initiale
— Les changements (certains au moins) doivent donc être anticipés durant la conception

74

Gestion des dépendances

— Les changements qui génèrent des problèmes sont les modifications inattendues dans les dépendances
— Ces dernières doivent donc être gérées durant la conception
— Les principes et les patterns de conception sont principalement liés aux dépendances

75

Principes et patterns de conception

— Principes SOLID
— Design Patterns
— Patterns GRASP
— KISS, YAGNI, DRY, Law of Demeter, ...

76

KISS : Keep It Simple, Stupid

KISS

« Simplicity should be a key goal in design and unnecessary complexity should be avoided », Kelly Johnson, ingénieur chez Lockheed

— Le code le plus simple est aussi le plus simple à maintenir

77

YAGNI : You Aren't Gonna Need It

- Ne pas ajouter une fonctionnalité avant que cela soit nécessaire
- Principe issu d'eXtreme Programming
  - « Do the Simplest Thing That Could Possibly Work »
- Nécessite de s'appuyer sur du *refactoring* pour être efficace

78

DRY : Don't Repeat Yourself

DRY

« every piece of knowledge must have a single, unambiguous, authoritative representation within a system », *The Pragmatic Programmer*, Andrew Hunt and David Thomas

- Chaque fonctionnalité doit être réalisée à un seul endroit du code
- Une modification d'un élément ne nécessite pas de changer un autre élément non relié logiquement

79

Law of Demeter

Law of Demeter or principle of least knowledge

- Each unit should have only limited knowledge about other units : only units "closely" related to the current unit.
- Each unit should only talk to its friends ; don't talk to strangers.
- Only talk to your immediate friends.

- Une méthode d'un objet devrait invoquer uniquement les méthodes de
  - l'objet lui-même,
  - ses paramètres,
  - tout objet qu'elle instancie,
  - ses composants directs.
- Un objet connaît le minimum de la structure de ses voisins
- Cela limite les dépendances avec les autres objets

80

Law of Demeter

Exemple

```
Wallet theWallet = theCustomer.getWallet();
double totalMoney = theWallet.getTotalMoney();
if (totalMoney > AMOUNT_TO_PAY_IN_EUROS) {
    theWallet.subtractMoney(AMOUNT_TO_PAY_IN_EUROS);
}
```

Listing 3.1 – Un client paye

- Cet exemple est extrait de The Paperboy, The Wallet, and The Law Of Demeter, David Bock
- Le créancier connaît la structure du client et manipule lui-même le portefeuille
- Il dispose de plus d'informations que nécessaire
- La validité du portefeuille n'est pas garantie

81

Law of Demeter

Exemple

```
public class Customer {
  private String name;
  private Wallet wallet;

  public Customer(String name, double fortune) {
    this.name = name;
    wallet = new Wallet(fortune);
  }

  public Wallet getWallet() {
    return wallet;
  }
}
```

Listing 3.2 – La classe client

82

Law of Demeter

Exemple

```
public class Wallet {
  private double totalMoney;

  public Wallet(double fortune) {
    totalMoney = fortune;
  }

  public double getTotalMoney() {
    return totalMoney;
  }

  public void subtractMoney(double amountToPayInEuros) {
    totalMoney -= amountToPayInEuros;
  }
}
```

Listing 3.3 – La classe portefeuille

83

Law of Demeter

Exemple

```
double paidAmount = theCustomer.getPayment(AMOUNT_TO_PAY_IN_EUROS);
```

Listing 3.4 – Un client paye (version corrigée)

- Le créancier doit demander le paiement
- La classe `Wallet` est isolée (diminue le couplage)
- La méthode `getPayment` encapsule la logique du paiement (améliore la cohésion)
- La classe `Customer` est plus complexe
  - mais la complexité a été transférée depuis le code de l'application

84

Law of Demeter

Exemple

```
public class Customer {
  private String name;
  private Wallet wallet;

  public Customer(String name, double fortune) {
    this.name = name;
    wallet = new Wallet(fortune);
  }

  public double getPayment(double amountToPayInEuros) {
    double totalMoney = wallet.getTotalMoney();
    double paidAmount = 0.0;
    if (totalMoney > amountToPayInEuros) {
      paidAmount = amountToPayInEuros;
      wallet.subtractMoney(paidAmount);
    }
   return paidAmount;
  }
}
```

Listing 3.5 – La classe client (version corrigée)

- La classe `Customer` ne publie plus la méthode `getWallet()`

85

3.2 Principes SOLID

3.2.1 Introduction

Principes SOLID

- Le premier jet de ces principes a été publié sur le newsgroup `comp.object` par Robert C. Martin en 1995

— Ces principes adressent la question de la gestion des dépendances dans la COO
— La bonne gestion des dépendances est nécessaire à la production d'un logiciel de qualité
— Les principes SOLID sont classés en trois groupes
  — cinq principes concernent la conception des classes (SOLID)
  — trois principes abordent la cohésion des modules
  — trois principes traitent du couplage entre modules

86

Principes liés à la conception des classes

SRP	Single Responsibility Principle	A class should have one, and only one, reason to change.
OCP	Open Closed Principle	You should be able to extend a classes behavior, without modifying it.
LSP	Liskov Substitution Principle	Derived classes must be substitutable for their base classes.
ISP	Interface Segregation Principle	Make fine grained interfaces that are client specific.
DIP	Dependency Inversion Principle	Depend on abstractions, not on concretions.

87

Principes liés à la cohésion des modules

REP	Release Reuse Equivalency Principle	The granule of reuse is the granule of release.
CCP	Common Closure Principle	Classes that change together are packaged together.
CRP	Common Reuse Principle	Classes that are used together are packaged together.

88

Principes liés au couplage entre modules

ADP	Acyclic Dependencies Principle	The dependency graph of packages must have no cycles.
SDP	Stable Dependencies Principle	Depend in the direction of stability.
SAP	Stable Abstractions Principle	Abstractness increases with stability.

89

3.2.2 Single Responsibility Principle (SRP)

Single Responsibility Principle (SRP)

Single Responsibility Principle (SRP)
A class should have only one reason to change.

— SRP est lié à la mesure de la *cohésion*
— La cohésion mesure le rapport entre une fonctionnalité attendue et le service rendu par une classe (ou un module)
— SRP relie cette notion au concept de changement
— Les exemples sont issus de SRP: The Single Responsibility Principle, Robert C. Martin

90

Exemple
La classe Rectangle viole SRP
(voir figure 3.1).

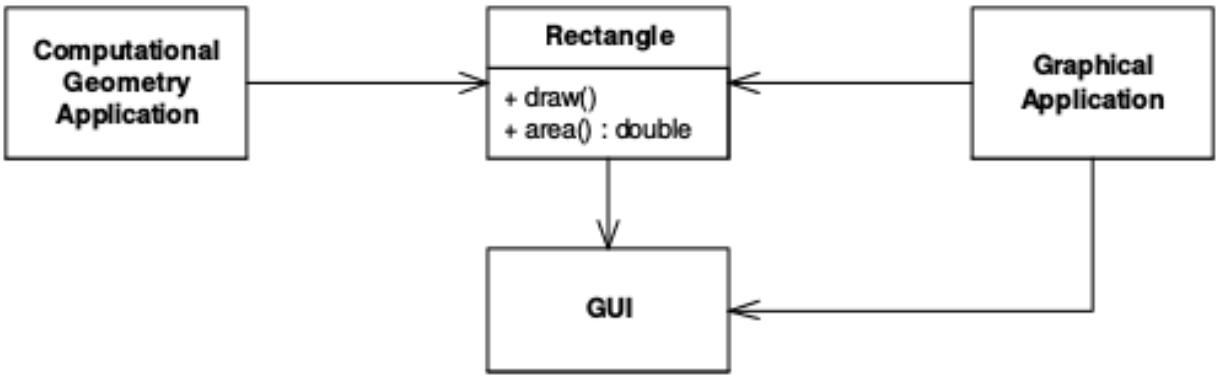

FIGURE 3.1 – Un exemple de violation de SRP.

— La classe `Rectangle` possède deux responsabilités
  — le calcul de surface,
  — l'affichage graphique.
— L'application de calcul géométrique dépend de l'affichage graphique
— Un changement de l'application graphique peut nécessiter un changement dans le rectangle et donc une reconstruction de l'application géométrique

91

Exemple
La classe Rectangle modfifée
(voir figure 3.2).

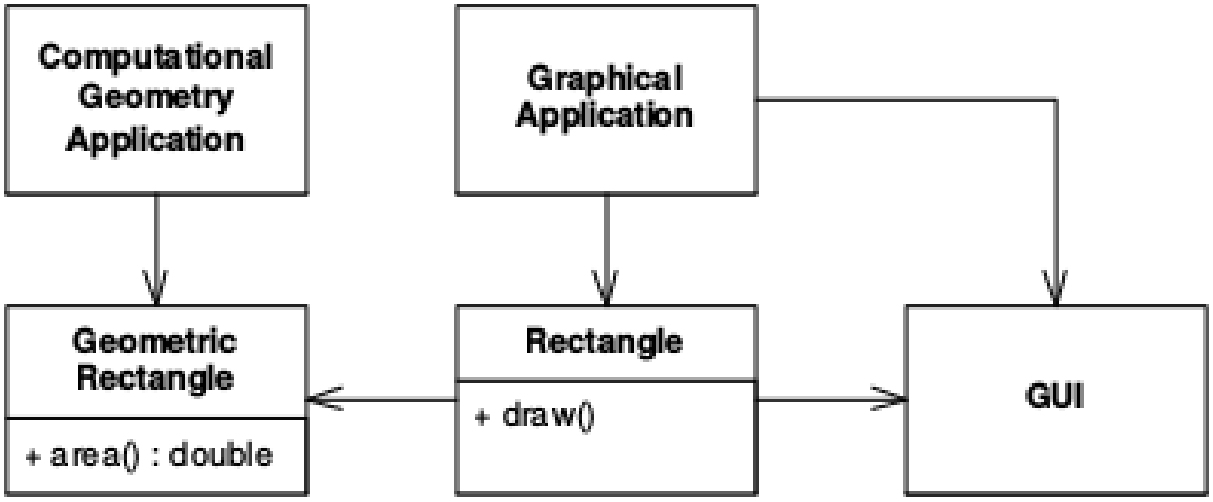

FIGURE 3.2 – Une application de SRP.

— Cette conception minimise l'impact des changements

92

Responsabilité
— Dans SRP, une *responsabilité* est définie comme « une cause de changement »
— Un changement dans les besoins provoquera une modification des responsabilités des classes
  ⇒ si une classe possède plusieurs responsabilités, elle aura plusieurs raisons de changer
— Si une classe possède plusieurs responsabilités, ces dernières sont couplées

⇒ un changement de l'une peut perturber le service de l'autre
— Quand n'est-il pas nécessaire de découpler les responsabilités ?
  — si les changements n'ont aucun risque de se produire
  — s'ils se produisent toujours ensemble

93

3.2.3 Open Closed Principle (OCP)

Open Closed Principle (OCP)

Open Closed Principle (OCP)
A module should be open for extension but closed for modification.

— Les modules (ou les classes) doivent pouvoir être étendus mais sans devoir être modifiés
  — le comportement doit pouvoir être changé sans modification du code source
— Les techniques permettant d'atteindre ce but sont basées sur l'*abstraction* en particulier sur les concepts OO

94

Intérêt et difficultés d'OCP
— L'intégration de nouveaux besoins ne nécessite que l'ajout de code et ne modifie pas l'existant
  — l'existant ne peut pas être dégradé
  — les modifications ne se propagent pas aux modules dépendants
— Il est difficile de respecter à la lettre ce principe
  — peut rendre la conception complexe
  — respecter OCP partiellement peut déjà apporter beaucoup à la conception
— Les exemples sont issus de The Open-Closed Principle, Robert C. Martin

95

Exemple
Un exemple de violation d'OCP 1/4

```
public enum ShapeType {
  CIRCLE, SQUARE;
}

public abstract class Shape {
  public final ShapeType type;

  public Shape(ShapeType type) {
    this.type = type;
  }
}
```

Listing 3.6 – La classe Shape

— L'attribut type représente le type de forme
— Suit une approche procédurale (non OO)

96

Exemple
Un exemple de violation d'OCP 2/4

```
public class Circle extends Shape {
  Point2D center;
  double radius;

  public Circle(Point2D center, double radius) {
    super(ShapeType.CIRCLE);
    this.center = center;
    this.radius = radius;
  }
}
```

Listing 3.7 – La classe Circle

97

Exemple
Un exemple de violation d'OCP 3/4

```
public class Square extends Shape {
  Point2D topLeft;
  final double side;

  public Square(Point2D topLeft, double side) {
    super(ShapeType.SQUARE);
    this.topLeft = topLeft;
    this.side = side;
  }
}
```

Listing 3.8 – La classe Square

98

Exemple
Un exemple de violation d'OCP 4/4

```
public static double computeArea(Shape s) {
  double result = 0;

  switch (s.type) {
  case CIRCLE:
    result = computeArea((Circle)s);
    break;
  case SQUARE:
    result = computeArea((Square)s);
    break;
  default:
    assert false : s.type;
  }
  return result;
}
```

Listing 3.9 – Le calcul de la surface

- Ne respecte pas OCP
  - l'ajout d'une forme oblige à modifier computeArea
- Ce motif se répétera dans toutes les fonctions qui devront différencier les formes

99

Exemple
Un exemple respectant OCP 1/3

```
public abstract class Shape {
    public abstract double computeArea();
}

public class Circle extends Shape {
    Point2D center;
    double radius;

    public Circle(Point2D center, double radius) {
        this.center = center;
        this.radius = radius;
    }

    @Override
    public double computeArea() {
        return PI * pow(radius, 2.0);
    }
}
```

Listing 3.10 – Les classes Shape et Circle

- Le type de forme s'appuie sur le polymorphisme

100

Exemple
Un exemple respectant OCP 2/3

```
public class Square extends Shape {
    Point2D topLeft;
    final double side;

    public Square(Point2D topLeft, double side) {
        this.topLeft = topLeft;
        this.side = side;
    }

    @Override
    public double computeArea() {
        return pow(side, 2.0);
    }
```

```
}
```

Listing 3.11 – La classe Square

101

Exemple

Un exemple respectant OCP 3/3

```
shapes = new ArrayList<>();
shapes.add(new Circle(new Point2D(1.0, 1.0), 1.0));
shapes.add(new Square(new Point2D(3.0, 4.0), 2.0));

double total = shapes.stream()
    .mapToDouble(Shape::computeArea)
    .sum();
```

Listing 3.12 – Le calcul de surface

— Le calcul utilise le polymorphisme
— Aucune modification de l'existant n'est nécessaire pour ajouter une forme

102

Implications d'OCP

— Les attributs doivent être privés
  — quand un attribut change, toute les méthodes qui en dépendent doivent changer aussi
  ⇒ ces méthodes ne sont pas fermées par rapport à cet attribut
    — normal pour les méthodes de la classe elle-même
    — non souhaitable pour les autres méthodes (*encapsulation*)
— Pas de variables globales
  — même argumentaire que pour les attributs mais par rapport aux modules
— Attention à l'usage des informations de typage à l'exécution (`instanceof` en java, `dynamic_cast` en C++, ...)
  — l'ajout d'un nouveau type peut provoquer un changement dans les méthodes

103

3.2.4 Liskov Substitution Principle (LSP)

Liskov Substitution Principle (LSP)

Liskov Substitution Principle (LSP)

Derived classes must be substitutable for their base classes.

— Est issu des travaux de Barbara Liskov
  — if for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T.
— également lié à l'approche *Design By Contract*, Bertrand Meyer
  — derived methods should expect no more and provide no less.
— Les exemples sont issus de The Liskov Substitution Principle, Robert C. Martin

104

Conséquence d'une violation du LSP

— Une méthode qui ne respecte pas LSP doit disposer d'informations sur les sous-classes
  ⇒ l'ajout d'un sous-classe impose de modifier la méthode
  ⇒ violation d'OCP

105

Exemple

Un exemple de violation de LSP 1/3

```
public class Rectangle {
    private int height;
    private int width;

    public int getHeight() { return height; }
    public void setHeight(int height) { this.height = height; }
    public int getWidth() { return width; }
    public void setWidth(int width) { this.width = width; }
}
```

Listing 3.13 – La classe Rectangle

106

Exemple

Un exemple de violation de LSP 2/3

```
public class Square extends Rectangle {
    private void setSide(int side) {
        super.setHeight(side);
        super.setWidth(side);
    }

    public void setHeight(int height) {
        setSide(height);
    }

    public void setWidth(int width) {
        setSide(width);
    }
}
```

Listing 3.14 – La classe Square

— Mathématiquement, un carré est un rectangle (relation *ISA*)
  ⇒ modélisé par un héritage entre Rectangle et Square
— Intuitivement, on sent que ce choix est discutable
  — height et width ne sont pas utile
  — idem pour les getters/setters correspondants
  — bricolage pour que le comportement soit adapté au carré

107

Exemple

Un exemple de violation de LSP 3/3

```
Rectangle r = new Rectangle();
r.setHeight(3);
r.setWidth(4);
assertThat(r.getHeight() * r.getWidth(), is(12)); // OK

Rectangle r = new Square();
r.setHeight(3);
r.setWidth(4);
assertThat(r.getHeight() * r.getWidth(), is(12)); // Échoue
```

Listing 3.15 – Méthode cliente

— L'utilisateur suppose que la modification de la hauteur n'a pas d'impact sur la largeur (et réciproquement)
— Le comportement n'est pas le même en présence d'un rectangle ou d'un carré
  ⇒ violation de LSP
  ⇒ le code client doit changer pour supporter la classe Square
  ⇒ violation d'OCP

108

Conséquence de LSP

— La validité d'un modèle n'est pas intrinsèque
  — dépend de son usage (des hypothèses de l'utilisateur du modèle)
  ⇒ lors de la conception, il faut « imaginer » ce que va supposer l'utilisateur
— La relation ISA porte sur le comportement
  — un carré est bien un rectangle d'un point de vue mathématique
  — un carré ne possède absolument pas le comportement d'un rectangle (indépendance entre hauteur et largeur)

— LSP et conception par contrat
  — l'utilisateur d'un objet d'une classe de base ne connaît que les pré et post-conditions de cette classe
  ⇒ toute sous-classe doit les respecter
  ⇒ la pré-condition ne peut pas être plus restrictive
  ⇒ la post-condition ne peut être que plus forte

109

3.2.5 Interface Segregation Principle (ISP)

Interface Segregation Principle (ISP)

Interface Segregation Principle (ISP)
Client should not be forced to depend upon interfaces that they do not use.

— ISP aborde la question de la taille des interfaces des classes
— Une interface ayant de trop nombreuses méthodes manque de cohésion
— Elle doit être découpée en fonction des besoins des clients
— Un client interagit à travers une interface adaptée à son besoin
— Les exemples sont issus de The Interface Segregation Principle, Robert C. Martin

110

Exemple
Un exemple de violation de ISP 1/3

```
public class Rectangle {
    private int height;
    private int width;

    public double computeArea() {
        return height * width;
    }

    public void draw(GraphicsContext gc) {
        gc.setFill(Color.GREEN);
        gc.fillRoundRect(110, 60, 30, 30, 10, 10);
    }
}
```

Listing 3.16 – La classe Rectangle

— fournit le calcul d'aire et l'affichage graphique (viole SRP)
— dépend de JavaFX

111

Exemple
Un exemple de violation de ISP 2/3

```
public class DrawingApp extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("Drawing Operations Test");
        Group root = new Group();
        Canvas canvas = new Canvas(300, 250);
        GraphicsContext gc = canvas.getGraphicsContext2D();

        Rectangle r = new Rectangle();
        r.draw(gc);

        root.getChildren().add(canvas);
        primaryStage.setScene(new Scene(root));
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 3.17 – L'application graphique

112

Exemple
Un exemple de violation de ISP 3/3

```
public class GeometricApp {
    public static void main(String[] args) {
        Rectangle r = new Rectangle();
        System.out.println(r.computeArea());
    }
}
```

Listing 3.18 – L'application géométrique

— Dépend inutilement de JavaFX
— Un changement dans les besoins de l'application graphique peut nécessiter une recompilation de l'application géométrique

Exemple
Un exemple respectant ISP 1/2

```
public interface DrawableRectangle {
    void draw(GraphicsContext gc);
}

public interface GeometricRectangle {
    double computeArea();
}
```

Listing 3.19 – Les interfaces

— Vont permettre d'isoler les applications de la classe concrète
— Met en œuvre le pattern ADAPTATEUR

Exemple
Un exemple respectant ISP 2/2

```
public class Rectangle implements GeometricRectangle, DrawableRectangle {
//...
```

Listing 3.20 – La classe Rectangle

```
DrawableRectangle dr = // logique de création de l'instance
dr.draw(gc);
```

Listing 3.21 – Dans l'application graphique

```
GeometricRectangle gr = // logique de création de l'instance
System.out.println(gr.computeArea());
```

Listing 3.22 – Dans l'application géométrique

— L'application graphique ne dépend plus de l'application graphique et de ses changements

3.2.6 Dependency Inversion Principle (DIP)

Dependency Inversion Principle (DIP)

Dependency Inversion Principle (DIP)

— High level modules should not depend upon low level modules. Both should depend upon abstractions.
— Abstractions should not depend upon details. Details should depend upon abstractions.

— Les abstractions sont moins sujettes aux changements
⇒ elles ne doivent pas dépendre d'éléments moins stables
— Plus de détails dans The Dependency Inversion Principle, Robert C. Martin

Exemple

Un exemple d'application de DIP

```
public static void copy(BufferedReader from, BufferedWriter to) throws IOException {
    String line = null;
    while ((line = from.readLine()) != null) {
        to.write(line);
    }
}
```

Listing 3.23 – Copie avec `java.io`

- `BufferedReader` et `BufferedWriter` sont des abstractions pour la source et la destination
- La logique de la copie est indépendante du type de source et de destination (fichier, mémoire, réseau, ...)
- La création d'instances viole souvent le DIP
  ⇒ utilisation du pattern AbstractFactory ou de l'*injection de dépendances*

117

3.2.7 Principes de cohésion des modules

Intérêt des modules

- Une classe possède un niveau de granularité trop fin pour une application de taille importante
- Il est nécessaire de disposer d'un outil de plus haut niveau pour organiser ce type d'application
- Les *modules* (ou *package*) représentent cet outil
  - ils permettent d'aborder la conception à un plus haut niveau d'abstraction
  - les classes sont partitionnées selon certains critères et placée dans des modules
  - les relations entre modules expriment l'organisation de haut niveau de l'application
- des questions de conceptions se posent donc vis à vis des modules
  - quels sont les meilleurs critères de partitionnement ?
  - quels liens existent entre les modules ?
  - quels principes gouvernent leurs conceptions ?

118

The Release/Reuse Equivalency Principle (REP)

The Release/Reuse Equivalency Principle (REP)

The granule of reuse is the granule of release.

- Un élément réutilisable peut être réutilisé uniquement s'il est géré par un système de distribution
- En particulier, il doit être associé à un numéro de version (de distribution)
- Un module est l'unité de distribution
  ⇒ un module est aussi l'unité de réutilisation
- Plus de détails dans Granularity, Robert C. Martin

119

The Common Reuse Principle (CRP)

The Common Reuse Principle (CRP)

Classes that aren't reused together should not be grouped together.

- Une dépendance avec un module est une dépendance avec tout ce qu'il contient
- Si une classe change, le module change et les clients doivent s'adapter
  ⇒ si une classe sans rapport mais se trouvant dans le module change, les clients sont impactés
  ⇒ les classes non utilisées ensemble ne doivent pas se trouver dans le même module
- Plus de détails dans Granularity, Robert C. Martin

120

The Common Closure Principle (CCP)

The Common Closure Principle (CCP)
Classes that change together, belong together.

- Une changement dans une classe implique une redistribution du module qui la contient
- On veut minimiser le nombre de modules qui changent entre deux versions de l'application
  ⇒ regrouper les classes qui changent ensemble permet d'atteindre ce but
- Plus de détails dans Granularity, Robert C. Martin

121

Combiner les trois principes

- Il est difficile de satisfaire les trois principes simultanément
- REP et CRP facilitent la réutilisation alors que CCP simplifie la maintenance
- CCP tend à produire de gros module alors que CRP en produit de petits
- On peut par exemple d'abord favoriser CCP pour la maintenance puis introduire REP et CRP quand le projet se stabilise

122

3.2.8 Principes liés au couplage entre modules

The Acyclic Dependencies Principle (ADP)

The Acyclic Dependencies Principle (ADP)
The dependencies betwen packages must not form cycles (DAG).

- Tous les modules se trouvant dans un cycle sont mutuellement dépendants
- Un cycle provoque une augmentation très importante du nombre de dépendances d'un projet
- Si un cycle a été ajouté dans un projet l'application des principes ISP et DIP permet de le supprimer
  - les interfaces sont isolées
  - les dépendances sont inversées
- Plus de détails dans Granularity, Robert C. Martin

123

The Stable Dependencies Principle (SDP)

The Stable Dependencies Principle (SDP)
Depend in the direction of stability.

- Un module dont dépendent de nombreux modules est difficile à changer
  ⇒ il est considéré comme très stable
- À l'inverse un module dont personne ne dépend est considéré comme instable, i.e. facile à changer
- Une application (et certains modules) doivent permettre les changements (par conception)
  ⇒ doivent être instables
  ⇒ aucuns modules stables ne doit dépendre d'eux
- Chaque module devraient uniquement dépendre de modules plus stables
- Plus de détails dans Stability, Robert C. Martin

124

The Stable Abstractions Principle (SAP)

The Stable Abstractions Principle (SAP)
Stable packages should be abstract packages.

- Les modules dont dépendent tous les autres sont difficiles à changer
- OCP impose qu'ils soient par contre extensible
  ⇒ ces modules doivent donc être très abstraits
- Une application est donc composée de

  - modules concrets instables (donc faciles à changer)
  - modules abstraits stables (donc faciles à étendre)
- La stabibilité d'un module doit être en accord avec sa stabilité
- SAP est donc directement lié à DIP
- Plus de détails dans Stability, Robert C. Martin

125

3.3 Patterns GRASP

3.3.1 Introduction

Patterns GRASP

- GRASP = General Responsibility Assignment Software Patterns
- Issus du travail de Craig Larman
- Ensemble de principes traitant de l'affectation de *responsabiblités* aux classes
- Tentative pour formaliser les intuitions utilisées par les concepteurs expérimentés
- Neuf patterns
  - Expert en information
  - Créateur
  - Faible couplage
  - Forte cohésion
  - Contrôleur
  - Polymorphisme
  - Fabrication pure
  - Indirection
  - Protection

126

Qu'est ce qu'une responsabiblité ?

- Une responsabilité correspond à une tâche qu'un objet ou un groupe d'objets doit réaliser
- Deux types : *Faire* et *Savoir*
- GRASP est un guide pour l'affectation de responsabilités aux objets

127

3.3.2 Expert en information

Expert en information

Problème
Érant donné un objet, quelles responsabilités peut-on lui attribuer ?

Solution
Lui sont assignées les responsabilités pour lesquelles il dispose des informations nécessaires à leur réalisation (Expert en information).

- Assez naturellement utilisé

128

3.3.3 Créateur

Créateur

Problème
Qui est responsable de créer une nouvelle instance d'un classe ?

Solution
Une classe B est responsable de créer une instance de A si

- B contient ou est composée de A, ou
- B enregistre A, ou
- B utilise A, ou
- B possède les données pour initialiser A.

— Limite le couplage
— Peut être insuffisant en cas de processus de création complexe (cf. Fabrique Abstraite)

129

3.3.4 Faible couplage

Faible couplage

Problème
Comment garantir un faible nombre de dépendances, limiter l'impact des changements et améliorer la réutilisation ?

Solution
Affecter les responsabilités de façon à maintenir un faible niveau de couplage.

— Peut être appliqué pour décider entre plusieurs alternatives
— Directement lié aux principes SOLID

130

3.3.5 Forte cohésion

Forte cohésion

Problème
Comment assurer que les objets sont compréhensible et maintenable ?

Solution
Assigner les responsabilités de façon à maintenir une forte cohésion.

— Peut être appliqué pour décider entre plusieurs alternatives
— Lié à Faible couplage

131

3.3.6 Contrôleur

Contrôleur

Problème
Comment gérer les interactions entre les messages systèmes (interface utilisateur, ...) et la couche métier ?

Solution
Assigner cette responsabilité à une classe parmi

le contrôleur façade représente le point d'entrée de l'ensemble du système

le contrôleur de session définit un point d'entrée par scénario/cas d'utilisation

— Le contrôleur *délègue* les traitements à la couche métier ou service
— Lié aux patterns d'architecture *MVC* (*Model-View-Controller*)
— Attention aux nombres de responsabilités affectées au contrôleur
  — trop de responsabilités ⇒ ajouter des contrôleurs
  — trop de traitements ⇒ déléguer

132

3.3.7 Polymorphisme

Polymorphisme

Problème
Comment gérer des alternatives basées sur le type ?

Solution
Assigner les responsabilités des comportements spécifiques aux classes dont le comportement est spécifique (grâce au polymorphisme)

— Passe en général par l'usage d'une classe abstraite ou une interface
— Les tests explicites sur le type dynamique d'un objet sont à proscrire
— Lié à OCP

133

3.3.8 Fabrication pure

Fabrication pure

Problème
Quel objet doit recevoir une responsabilité en assurant Forte cohésion, Couplage faible quand les autres principes sont inappropriés ?

Solution
Créer une classe artificielle (ne représentant pas un concept du domaine) et lui affecter cette responsabilité.

— Toutes les classes d'une application ne sont pas des concepts métiers
— Concerne en particulier les fonctionnalités techniques (persistance, logging, ...)

134

3.3.9 Indirection

Indirection

Problème
Comment assigner une responsabilité en évitant le couplage direct entre des objets ?

Solution
Assigner la responsabilité à un objet intermédiaire qui assure la médiation.

— L'intermédiaire est l'Indirection

135

3.3.10 Protection

Protection

Problème
Comment concevoir les éléments d'un système de telle façon que les variations de ces éléments n'aient pas d'effets indésirables sur les autres ?

Solution
Anticiper les points de variations (ou d'instabilité) et créer un interface stable autour d'eux.

— Les principes SOLID détaillent ce principe

136

3.4 Design patterns

3.4.1 Introduction

Patron de conception

- Un *patron de conception* (*Design pattern*) est une solution générique d'implémentation répondant à un problème spécifique.
- Communément utilisés dans un contexte OO sous la forme d'une structure de classe.
- Popularisés par le livre **Design Patterns : Catalogue de modèles de conceptions réutilisables**, *Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides*, Vuibert, 1999
  - généralement nommé *GoF* (*Gang of Four*)
  - présente 23 patterns et les appliques sur des exemples en C++
- Un pattern est la formalisation de bonnes pratiques communément utilisées pour la résolution d'un problème récurrent
  - l'expression d'un pattern est composée d'un ensemble de section (nom, objectif, applicabilité, structure, ...)

137

Utilisation des design patterns

- Les patrons de conception sont des outils supplémentaires pour réaliser une bonne conception
  ⇒ s'intègre naturellement dans un processus de développement
  - La difficulté réside dans l'identification des situations d'application
- Chaque pattern doit être instancié dans le contexte où il est utilisé
- Ils ne sont pas toujours une bonne solution (risque de sur-conception)

138

Notions liées

Idiome construction utilisé de façon récurrente dans un langage de programmation donné pour réaliser une tâche « simple »
- `i++` à la place de `i = i + 1` en Java, C, ...
- `for (element : liste) {}` pour parcourir les éléments d'une collection

Pattern d'architecture solution générique et réutilisable à un problème d'architecture logicielle
- modèle *MVC* (*Model-View-Controller*)

Pattern d'entreprise solution pour la structuration d'une application d'entreprise
- Service Layer

Anti-pattern solution commune à un problème récurrent mais qui est en général inefficace et contre-productive
- Anemic domain model
- God object

139

Classification des design patterns

Patrons de création traitent de la création et de l'initialisation d'objet
- masque les classes concrètes utilisées et leur mode de création

Patrons de structure traitent de l'organisation des relations entre classes

Patrons de comportement traitent de la communication entre objets

140

3.4.2 Patrons de création

Titre

-

141

3.4.3 Patrons de structure

Titre

—

142

3.4.4 Patrons de comportement

Titre

—

143

3.5 Exercices

CONTRAINTES

- L'ensemble des exercices de ce module doit être réalisé dans un projet dont la compilation est assurée par MAVEN et la gestion de versions par GIT.
- Chaque étudiant doit se créer un compte GITHUB avec un login de la forme uvsqXXXXX où XXXXX est le numéro d'étudiant. Le projet avec les exercices doit se trouver sur ce compte.
- Le projet MAVEN doit permettre les actions suivantes :
  - `mvn package` pour produire un `jar` du projet
  - `mvn exec:java` pour exécuter
- Le code source du projet doit être placé dans le package `fr.uvsq.coo`. Chaque exercice doit se trouver dans son propre package `fr.uvsq.coo.exC_EE` où `C` est le chapitre et `EE` le numéro de l'exercice (par exemple *Exercice 3.1* se trouve dans le package `fr.uvsq.coo.ex3_1`).

Exercice 3.1 (Mise en place du projet)
Cet exercice a pour seul but de vous aider à initialiser le projet que vous utiliserez tout au long des exercices.

- Sous GITHUB, créez un nouveau dépôt pour le projet (avec un `README` et un `.gitignore` pour MAVEN) (cf. GitHub Bootcamp).
- Clonez localement ce projet avec GIT.
- Générez le squelette du projet MAVEN à l'aide de l'archetype `maven-archetype-quickstart` (cf. Maven in 5 Minutes).
- Vérifiez que le projet se construit correctement.
- Validez les changements et synchroniser avec le dépôt GITHUB.
- Mettez à jour la dépendance avec JUNIT vers la version la plus récente (cf. MVNRepository) et modifier le code de test en conséquence (cf. A Simple Unit Test).
- Vérifiez que le projet se construit correctement, validez et synchronisez.

Exercice 3.2 (The Bowling Game Kata)
Dans cet exercice, vous vous familiariserez avec le développement dirigé par les tests (*Test Driven Development* ou `TDD`). Pour cela, vous suivrez les étapes du document The Bowling Game Kata, Robert C. Martin. Une copie au format PDF du document est disponible sur e-campus.

- Effectuez les différentes étapes du document.
- Adaptez les tests unitaires à la version de JUNIT du projet.
- Validez les changements après chaque test réussi.
- À la fin, n'oubliez pas de synchroniser le projet sur GITHUB.

Exercice 3.3 (Illustration du principe de responsabilité unique (SRP))
Soit la classe `Employe`

```java
class Employe {
    private final String nom;
    private final String adresse;

    // ...

    public double calculSalaire() { return /* calcul du salaire */; }
    public void afficheCoordonnees() { System.out.println(nom + ", " + adresse); }
}
```

1. Cette classe respecte-t-elle SRP ? Pourquoi ?
2. Que se passe-t-il si la méthode de calcul du salaire change ?
3. Que se passe-t-il si l'affichage est remplacé par le stockage dans un fichier CSV ?
4. Proposez une solution respectant SRP.

Exercice 3.4 (Illustration du principe ouvert/fermé (OCP))
Le salaire d'un employé est de 1500€ auquel s'ajoute 20€ par année d'ancienneté. Le salaire d'un vendeur se calcule sur la même base mais en ajoutant une commission propre à chaque vendeur. On veut pouvoir calculer la somme totale des salaires de l'entreprise.

1. Proposez une solution respectant OCP.
2. Pour le vérifier, ajoutez la classe manager (même base de calcul que l'employé plus 100€ par personne sous ses ordres).

Exercice 3.5 (Illustration du principe de substitution de Liskov (LSP))
Soient les classes `Robot` et `RobotStatique`

```java
class Robot {
    private Position position;
    private Direction direction;

    public void tourne() { /* tourne d'1/4 de tour */ }
    public void avance() { /* avance d'une case */ }
}

class RobotStatique {
    @Override
    public void avance() { thrown new UnsupportedOperationException(); }
}
```

1. Cette solution respecte-t-elle LSP ? Pourquoi ?
2. Implémentez la méthode `avancerTous` qui fait avancer tous les robots.
3. Proposez une solution respectant LSP.

Exercice 3.6 (Illustration du principe de ségrégation des interfaces (ISP))
Soit le code Java suivant :

```java
interface Printer {
    void print();
    void scan();
    void copy();
    void fax();
}

class SimplePrinter implements Printer {
    @Override
    public print() { /* print a document */ }

    @Override
    void scan() { thrown new UnsupportedOperationException(); }

    @Override
    void copy() { thrown new UnsupportedOperationException(); }

    @Override
    void fax() { thrown new UnsupportedOperationException(); }
}
```

1. Quels problèmes peuvent se poser avec cette solution ?
2. Supposons qu'une application utilisant le fac nécessite de changer l'interface de la méthode `fax` en `void fax(List<Document> l);`, quel impact cela aura-t-il sur `SimplePrinter` ?
3. Proposez une solution respectant ISP.

Exercice 3.7 (Illustration du principe d'inversion des dépendances (DIP))
Soit le code Java suivant :

```
class UneClasseMetier {
    public void uneMethodeMetier() {
        System.out.println(LocalDateTime.now() + ": Début de uneMethodeMetier"); // log message

        // Traitements métiers

        System.out.println(LocalDateTime.now() + ": Fin de uneMethodeMetier"); // log message
    }
}
```

1. Ce code respecte-t-il DIP ? Pourquoi ?
2. Proposez une solution respectant DIP.

Chapitre 4

Conclusion

Sommaire

4.1 Début de la conclusion

Bilan

1. blabla

144

Pour aller plus loin...

1. blabla

145