Jacques Lonchamp

Conception d'applications en Java/JEE

Principes, *patterns* et architectures

DUNOD

Illustration de couverture :
Abstract triangle mosaic background with dotted line structure © karandaev - Fotolia.com

5 rue Laromiguière, 75005 Paris
www.dunod.com

ISBN 978-2-10-071686-9

Table des matières

Avant-propos

POURQUOI CET OUVRAGE ?

Cet ouvrage est l'aboutissement d'une longue pratique de l'enseignement du développement logiciel. De cette pratique ont émergé quelques convictions fortes concernant la pédagogie de l'informatique. Les insatisfactions éprouvées à la lecture de la littérature traitant de la conception logicielle, à l'occasion d'une réflexion nationale sur la formation des informaticiens, ont constitué l'élément déclencheur de sa rédaction.

Positionnement dans le cursus

La première conviction est que tout enseignement de l'informatique comporte trois phases qui obéissent à des finalités bien distinctes. Les ouvrages à vocation pédagogique doivent se positionner clairement dans ce processus.

La phase « disciplinaire »

Elle vise l'acquisition des savoirs conceptuels et techniques de base, grâce à des enseignements ciblés vers des domaines bien délimités. Les documents pédagogiques correspondants ne doivent exiger aucun prérequis.

Pour ce qui est du développement logiciel, on trouve le plus souvent dans ce socle de base, des enseignements en algorithmique et structures de données, en programmation objet, en programmation Web, en bases de données, en UML et en interfaces homme-machine. La littérature correspondante est extrêmement riche.

La phase « intégrative »

Elle est indispensable pour tisser des liens entre les savoirs disciplinaires, les approfondir en conséquence, et permettre une prise de conscience des enjeux dans le monde professionnel.

En ce qui concerne le développement logiciel, cette deuxième phase est souvent organisée autour des thématiques de l'*analyse* – des problèmes et des besoins –, de la *conception* – architecturale et détaillée – et des *processus et environnements de développement* – incluant la gestion des projet. Cet ouvrage concerne le deuxième de ces thèmes intégratifs, « *la conception objet* ».

Les insuffisances de la littérature pédagogique à ce niveau seront détaillées dans la suite.

La phase « professionnalisante »

Elle s'appuie sur la compréhension générale des grandes thématiques du développement logiciel que procure la phase intégrative. Elle vise l'approfondissement de thèmes techniques spécialisés permettant de passer de la *compréhension* à la *maîtrise effective* d'une des facettes au moins du développement logiciel. Il peut s'agir, par exemple, de l'approfondissement des technologies JSF, JPA et EJB, pour former des développeurs d'applications JEE.

Les documents pédagogiques correspondants peuvent faire l'hypothèse d'une maîtrise suffisante des fondements conceptuels et du contexte professionnel. Les ouvrages techniques professionnels servent souvent de support pédagogique pour cette phase.

Importance des ancrages explicites

La seconde conviction est qu'un apprentissage est d'autant plus efficace qu'il est *explicitement ancré* dans ce qui a déjà été acquis par l'apprenant. Par exemple, un patron de conception sera d'autant plus convainquant et facile à mémoriser qu'on montrera qu'il a déjà été utilisé sans le savoir dans telle ou telle construction du langage de programmation pratiqué jusque-là.

Dans cet esprit, la première partie de cet ouvrage s'appuie systématiquement sur le langage Java et l'organisation du JDK pour illustrer et justifier les patrons de conception. De même, la discussion des principes abstraits de conception est elle-même ancrée dans la réalité plus « tangible » des patrons de conception, qui sont présentés en premier.

Insuffisances de l'offre pédagogique

La première insatisfaction à la lecture de la littérature existante réside dans le faible nombre de documents abordant de manière synthétique le thème de la conception logicielle, y compris en langue anglaise. La majorité des syllabus se contentent de citer les ouvrages professionnels de référence sur les patrons de conception (le « GoF » [Gam+95]) et sur les principes de conception proposés par les principaux « gourous » du domaine, comme Robert Martin – « Uncle Bob » [Mar00 ; Mar02] et Martin Fowler [Fow02].

Parmi les quelques exceptions, on peut citer l'ouvrage *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* [Lar95]), très pédagogique et à spectre large, et l'ouvrage *Design Patterns – Tête la première* [Fre+04]

qui étend aux patrons de conception l'approche pédagogique « à l'américaine » des autres ouvrages de la collection « Tête la première », avec quiz et cartoons à foison.

Caractère trop parcellaire des ouvrages existants

La deuxième insatisfaction réside dans le caractère trop *parcellaire* des ouvrages abordant la conception logicielle. Il y est question de patrons de conception, de principes de conception ou d'architectures logicielles, mais rarement des trois à la fois. Il y est question d'applications « classiques » pour le poste de travail, d'applications Web élémentaires ou d'applications multi-tiers, mais rarement des trois à la fois. Ce caractère parcellaire est aux antipodes de ce qu'on peut attendre d'un ouvrage pédagogique sur un thème « intégratif ».

Caractère trop abstrait des ouvrages existants

La troisième insatisfaction réside dans la *faiblesse des exemples, études de cas et exercices d'application proposés*. Bien qu'ouvrage de référence reconnu et remarquable, le « GoF » est illisible pour la majorité des étudiants actuels à cause de son organisation plutôt indigeste (présentation des 23 patrons selon une grille rigide en 13 sections), à cause de ses exemples (uniquement en C++ et Smalltalk, très centrés sur les frameworks graphiques de l'époque) et à cause de l'absence de tout exercice d'application. Or les illustrations et exercices sont essentiels du point de vue pédagogique. Une faiblesse dans ce domaine peut renforcer l'idée qu'il ne s'agit que d'un « *discours théorique* » dont on peut faire l'économie.

Au contraire, le présent ouvrage présente systématiquement plusieurs exemples par thème abordé (exemples simplement « illustratifs » du thème et exemples « de conception », impliquant une certaine réflexion sur le thème), une soixantaine d'exercices d'application tous corrigés, et deux études de cas (JSE et JEE) détaillées jusqu'au code complet. Les codes des études de cas sont disponibles sur *www.dunod.com/contenus-complementaires/9782100716869*. La majorité des exercices ont été glanés sur le Web et plus ou moins retravaillés. Que leurs auteurs originaux, souvent impossibles à déterminer, soient ici remerciés collectivement.

POUR QUELS LECTEURS ?

Cet ouvrage pédagogique s'adresse prioritairement aux étudiants de deuxième et troisième année des cursus spécialisés en informatique, quelle que soit leur nature (DUT/LP, L2/L3, deuxième et troisième années d'écoles d'ingénieurs).

Cet ouvrage peut bien entendu être également utile à tous les praticiens de l'informatique qui souhaitent rafraîchir ou élargir leurs connaissances en conception logicielle.

Soulignons enfin que les connaissances en JEE étant très variables selon les cursus, et parfois inexistantes, tous les concepts essentiels sont rappelés de manière synthétique en début d'ouvrage, en mettant l'accent sur la compréhension des composants et de leurs interrelations.

Chapitre 1

Introduction

1.1 DÉFINITIONS

La phase de *conception logicielle* est l'équivalent, dans le domaine de l'informatique, de la phase de conception que l'on retrouve dans toutes les disciplines de l'ingénierie traditionnelle (génie mécanique, génie civil, génie électrique, etc.). Toute conception a pour finalité *de penser et de représenter le produit à réaliser sous une forme abstraite, avant sa production effective*. Cependant, la nature immatérielle du logiciel rend la frontière entre la conception et le produit plus floue que dans l'ingénierie traditionnelle. La différence entre le plan d'un pont et le pont lui-même est plus évidente que la différence entre un schéma de classes et le programme objet qui l'implante.

La *conception logicielle orientée objet* est la discipline qui s'intéresse à la construction des *modèles de conception objet*. Ces modèles sont la représentation abstraite, le plus souvent à l'aide d'UML, de l'ensemble des objets logiciels et de leurs interactions permettant de résoudre un problème et de satisfaire les besoins identifiés et décrits durant la phase précédente d'*analyse orientée objet*. Une fois le modèle de conception établi, les développeurs peuvent l'implanter à l'aide d'un *langage de programmation orienté objet*.

De manière schématique, on peut dire que l'analyse répond aux questions : *Quel est le problème ? Quels sont les besoins ?* Alors que la conception répond aux questions : *Comment résout-on le problème ? Comment satisfait-on les besoins ?*

Les entrées de la conception orientée objet sont donc les sorties de l'analyse orientée objet. Elles consistent en général en : (1) un modèle des *concepts du domaine*, le plus souvent sous la forme d'un diagramme de classes UML, (2) la description des principaux *services*

(fonctionnalités) que le futur système devra procurer à ses différentes catégories d'utilisateurs, le plus souvent sous la forme de cas d'utilisation UML, accompagnée parfois de scénarios d'utilisation, (3) éventuellement, la spécification des *caractéristiques non fonctionnelles* qu'il faut satisfaire (efficacité, sûreté, disponibilité, etc.), (4) éventuellement, l'esquisse de l'*interface utilisateur* qui sera offerte.

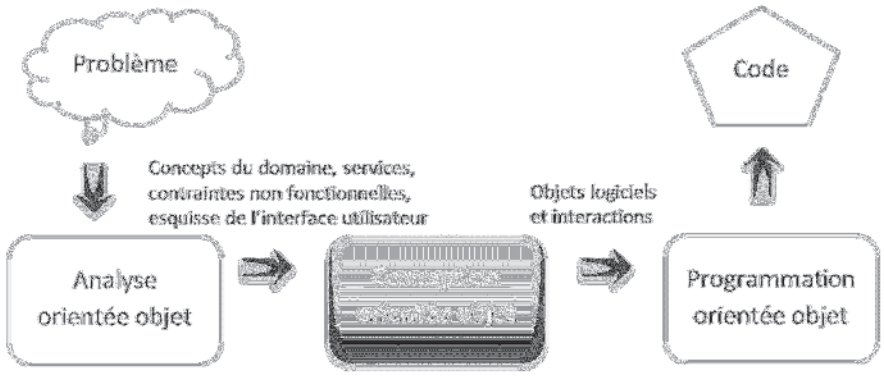


FIGURE 1.1 Place de la conception orientée objet

Analyse, conception, programmation, test, livraison, maintenance, ne doivent pas être compris nécessairement comme des étapes successives, exécutées séquentiellement. Cette vision correspond au « modèle de la cascade » [Roy70] des débuts de l'informatique dans laquelle l'analyse était faite de manière complète au début du processus (ce qui impliquait des besoins clairs et stables et un domaine d'application complètement maîtrisé) et l'application était testée et livrée en une seule fois à la fin de ce (long) processus.

Aujourd'hui, les applications sont le plus souvent produites par des processus *itératifs et incrémentaux*. « Incrémental » signifie que des parties sont développées au fil du temps, de manière planifiée, et intégrées dès qu'elles sont terminées. « Itératif » signifie qu'une partie est remaniée et améliorée plusieurs fois avant d'atteindre son état définitif.

Les « *méthodes agiles* » [Bec99] sont la concrétisation actuelle de ces idées. Elles sont caractérisées par des itérations très courtes (quelques semaines au plus), planifiées de manière souple. Ces itérations correspondent à la fois à des remaniements et à des ajouts. Ces approches permettent de livrer rapidement et régulièrement des versions utilisables de plus en plus complètes et de s'adapter à toutes les formes de changement des besoins ou du domaine, même quand elles apparaissent tardivement.

1.2 OBJECTIFS

La conception orientée objet est un art difficile. Dans le cadre des approches à base de classes, le modèle de conception objet doit décrire les objets et classes nécessaires, leurs interfaces, leurs regroupements en paquets, ainsi que les relations que ces éléments entretiennent, en particulier en termes de dépendances. Ce point sera détaillé au paragraphe suivant. Dans un

contexte JEE, les classes correspondent à beaucoup de concepts différents (servlet, JSP, Java Bean, EJB, entité JPA, etc.) qu'il faut maîtriser ainsi que les dépendances spécifiques qu'ils impliquent.

La conception doit à la fois être *spécifique* au problème à résoudre, qu'il faut avoir minutieusement analysé, et parfois un peu plus *générale*, pour faciliter l'adaptation aux évolutions qui se produiront inévitablement.

Les objectifs d'une « bonne conception » sont multiples. Elle doit être facile à comprendre (*intelligibilité*). La modification des fonctionnalités existantes doit être aussi aisée que possible, sans reprogrammations excessives (*flexibilité*), de même que l'implantation de nouvelles fonctionnalités (*extensibilité*). Ces évolutions ne doivent pas mettre en péril l'existant (*robustesse*, non-régression). Certaines parties d'une application doivent pouvoir être reprises pour en construire d'autres (*modularité et réutilisabilité*).

1.3 PROBLÉMATIQUE

Même quand on part d'une « bonne conception », une longue suite de modifications, inévitables et souvent faites dans l'urgence par d'autres que les concepteurs initiaux, conduit souvent à une forte dégradation de celle-ci (« effet spaghetti » [Mar00]).

Une condition fondamentale pour satisfaire les objectifs précédents sur la durée est de bien maîtriser *la nature et le nombre des dépendances* entre les éléments, classes et paquets, des applications. On parle de dépendance entre un élément A et un élément B, notée A $\rightarrow$ B, quand le fonctionnement de l'élément A *requiert la présence* de l'élément B, qui *joue un certain rôle* dans ce fonctionnement. En conséquence, tout changement apporté à la partie visible (publique) de B peut impacter aussi A. En outre, A ne peut pas être utilisé dans un contexte autre que celui de B.

Une classe A dépend d'une classe B si au moins une des conditions suivantes est vérifiée : A possède un attribut de type B (dépendance par composition), A est de type B (dépendance par héritage), A dépend de la classe C qui dépend de la classe B (dépendance par transitivité), une méthode de A appelle une méthode de B. Les dépendances entre paquets résultent des dépendances entre les classes qu'ils contiennent, quelle que soit leur nature.

On distingue les dépendances *directes* (A $\rightarrow$ B), *transitives*, quand il existe un ou des éléments intermédiaires (A $\rightarrow$ C $\rightarrow$ B), *cycliques*, quand la transitivité crée une « boucle » d'un élément vers lui-même (A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D $\rightarrow$ A). Les dépendances transitives sont souvent considérées comme moins importantes que les dépendances directes, car la majorité des changements ne se propagent pas à travers ces dépendances transitives [Jac03]. Elles rendent cependant plus difficile la *compréhension* de l'application (pour comprendre A il faut comprendre tous les éléments qui lui sont liés directement ou indirectement), la *réutilisation* de ses éléments (A doit être accompagné de tous les éléments liés directement ou indirectement), le *test* de l'application (le test de A nécessite la présence ou la simulation de tous les éléments liés directement ou indirectement). Les dépendances cycliques doivent être évitées absolument, en particulier pour les paquets [Fow01]. Si A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ A, les trois paquets A, B et C ne peuvent plus être utilisés, testés, et versionnés séparément.

Les qualités attendues d'une bonne conception, comme l'intelligibilité, l'extensibilité, la flexibilité et la réutilisabilité sont très liées au nombre et à la nature des dépendances entre les

éléments des applications. La minimisation des dépendances facilite la gestion des changements, autrement dit la maintenance des applications, puisqu'il y a moins d'éléments susceptibles d'être impactés par ces changements. Au contraire, plus il y a de dépendances et plus le travail des développeurs est rendu difficile !

1.4 CONTENUS ET PROGRESSION

Prérequis

La maîtrise de la conception orientée objet nécessite des connaissances et compétences de base en programmation orientée objet et en modélisation de la structure et du fonctionnement des applications à l'aide d'UML. Le minimum à connaître dans ces domaines est rappelé dans les deux chapitres de la partie de rappels de cours, intitulés « Java/JEE » et « UML ».

Approche

La maîtrise de ces concepts de base, si elle est importante, *ne suffit pas à guider la conception des applications objet*. Les connaissances spécifiques à acquérir en conception orientée objet prennent diverses formes : des *principes de conception théoriques*, des *patrons de conception*, répondant *concrètement* à des problèmes récurrents rencontrés en conception orientée objet, des *styles et patrons d'architecture*, pour les différents types d'applications à base d'objets. Toutes ces connaissances expriment, sous des formes plus ou moins opérationnelles, la manière de bien gérer les dépendances entre éléments des applications.

Il a souvent été suggéré une analogie entre apprentissage de la conception et apprentissage du jeu d'échecs. Pour progresser dans le jeu d'échecs, il faut analyser les parties des grands maîtres et mémoriser leurs caractéristiques afin de pouvoir les réutiliser. De manière similaire, pour progresser en conception, il faut étudier les pratiques éprouvées, recueillies et formalisées par les spécialistes reconnus du domaine, les mémoriser et les réutiliser.

Dans leurs fondements, ces connaissances ne dépendent pas d'un langage de programmation orienté objet particulier. Cependant, dans l'implantation pratique des idées, le langage de programmation joue un rôle qui ne peut être ignoré. Java est l'unique langage de programmation cible considéré dans cet ouvrage.

Progression pédagogique

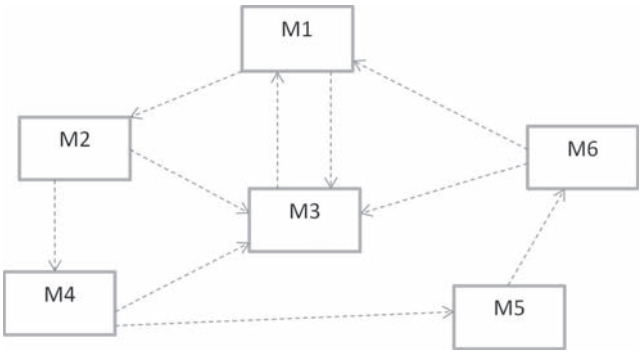
Le domaine de la conception orientée objet est très vaste. Cet ouvrage en fait une présentation synthétique qui préserve cependant la multiplicité des points de vue évoquée ci-dessus : principes théoriques, patrons de conception concrets, styles et patrons architecturaux.

La progression pédagogique qui est proposée *part de la programmation*. Les patrons de conception principaux sont présentés *comme des choses déjà rencontrées durant l'apprentissage de Java*, ce qui les rend plus concrets et plus faciles à mémoriser. Puis, les principes abstraits sont introduits, à chaque fois que possible, comme des *généralisations* de certains patrons de conception. Puis, les styles et patrons architecturaux sont décrits comme des *concrétisations* des principes généraux, et parfois, comme des *combinaisons* de patrons de conception. Certaines descriptions restent au niveau de la simple « culture générale ». Au

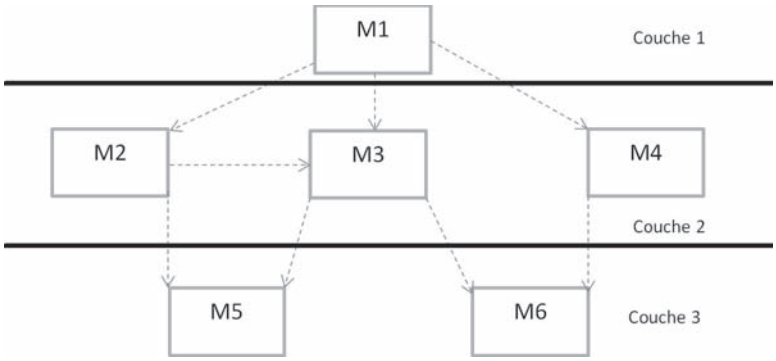
contraire, en raison de leur importance actuelle, un éclairage tout particulier est mis sur les architectures Web, trois-tiers et multi-tiers, dans le contexte JEE. Les patrons spécifiques à l'utilisation de cette plateforme sont explicités. Des illustrations concrètes de mise en œuvre sont fournies. Enfin, le développement de *compétences pratiques* est abordé à travers deux études de cas non triviales : la conception et la programmation en Java SE d'un moteur générique de jeu d'arcade 2D et la programmation en Java EE d'une application de commerce électronique.

EXERCICES

Exercice 1.1. Soit l'architecture en modules de la figure suivante, où les flèches en pointillé décrivent des dépendances interpaquets. Quel est l'impact potentiel d'une modification de chaque module ?



Exercice 1.2. (suite de l'exercice précédent)
Après réingéniérie du code, on a obtenu l'architecture en trois couches de la figure suivante. Quel est l'impact potentiel d'une modification de chaque module dans cette nouvelle architecture ?



PARTIE 1

Les rappels de cours

Chapitre **2**

Java/JEE

2.1 MODULARITÉ ET ENCAPSULATION

La stratégie de base de toute conception consiste à décomposer le système considéré en un assemblage de sous-systèmes, ou « modules », plus simples.

Dans une bonne conception, le *couplage* des modules, c'est-à-dire le nombre de dépendances entre eux, doit rester faible. Comme souligné dans l'introduction, un faible nombre de dépendances minimise l'impact des changements et facilite la réutilisation.

De manière complémentaire, la *cohésion* interne des modules doit être forte. Un module doit regrouper des éléments qui ne sont pas disparates. Par exemple, dans une application de jeu, un module qui fait évoluer le jeu en même temps qu'il affiche des éléments à l'écran présente une faible cohésion interne. Il faudrait envisager de le scinder en un module d'affichage et un module de traitement. Une forte cohésion facilite l'intelligibilité et la réutilisabilité.

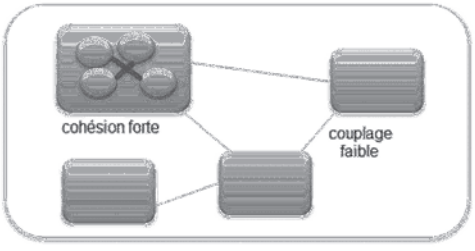


FIGURE 2.1 Couplage et cohésion

L'*encapsulation* est une technique qui favorise la modularité. Elle consiste à séparer l'*interface* du module, c'est-à-dire la liste des services qu'il offre – correspondant à un type abstrait – et son *implantation*, c'est-à-dire la réalisation de ses services, par des données et des algorithmes.

Toutes les interactions se font via l'interface, ce qui évite le couplage par le contenu des modules. L'implantation (privée) est cachée et peut donc évoluer sans conséquences sur les autres modules tant que l'interface publique reste inchangée. Et si l'interface publique évolue, le couplage faible en limite les conséquences. L'encapsulation permet également d'avoir plusieurs versions de l'implantation pour la même interface.

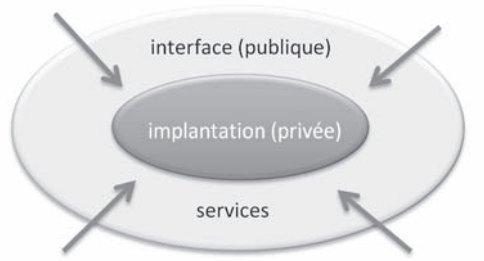


FIGURE 2.2 Public et privé

2.2 OBJET

En Java un *objet* est un module qui possède une identité unique et invariable. Il contient des données – son état, constitué par les valeurs de ses *variables d'instance* – et des *méthodes*, susceptibles de modifier son état et qui peuvent éventuellement retourner des résultats.

Les méthodes sont plutôt publiques et les variables d'instance plutôt privées. En réalité, c'est au choix du programmeur, via les modificateurs d'accessibilité `public` et `private`. Dans cette configuration préférentielle, les interactions entre objets se font via l'invocation des méthodes. L'accès aux variables d'instance se fait par des accesseurs (*getters*) et leur mise à jour, par des mutateurs (*setters*).

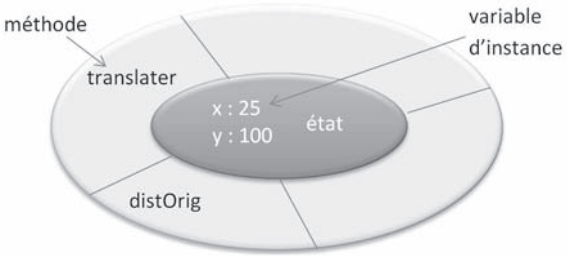


FIGURE 2.3 Un objet point

2.3 CLASSE

Une *classe* Java est un générateur d'objets de même structure (variables d'instance) et de même comportement (méthodes). La création d'un objet, son instanciation, se fait par l'opérateur `new` qui fait appel à un constructeur.

Exemple : la classe `Point`

```
public class Point {
  // variables d'instance privées
  private int x;
  private int y;
  // constructeur
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  // méthodes
  public void translater(int dx, int dy) {
    x = x + dx;
    y = y + dy;
  }
  public double distOrig() {     // distance à l'origine
    return Math.sqrt(x*x + y*y);
  }
  public int getX() {return x;}
  public int getY() {return y;}
}
Point unPoint;                  // déclaration d'une variable (type déclaré)
unPoint = new Point(25, 100);   // instanciation (type réel de l'objet créé)
unPoint.translater(10, 10);     // utilisation (appel de méthode sur l'objet)
```

Le *type déclaré* (ou type statique) est le seul connu par le compilateur. Il correspond à l'interface utilisable, c'est-à-dire à l'ensemble des méthodes que l'on peut appeler.

Le *type réel* pris lors de l'exécution (ou type dynamique), connu par la machine virtuelle, correspond à la classe utilisée lors du `new`.

Pour tout appel de méthode le compilateur vérifie que la méthode existe bien dans le type déclaré. La machine virtuelle, quant à elle, appelle la méthode correspondant au type réel de l'objet. On verra dans la suite que le type réel peut être différent du type déclaré.

2.4 HÉRITAGE

2.4.1. Définition

L'héritage entre classes est un moyen pour réutiliser du code. Mais ce n'est pas le *seul* moyen et pas toujours le *bon* moyen. . . , comme expliqué dans la suite.

L'héritage consiste à définir une classe, la « sous-classe » ou « classe dérivée », à partir d'une autre classe, la « superclasse ». La sous-classe *hérite* de tous les attributs et de toutes les

méthodes de la superclasse et peut en ajouter d'autres. Seul l'héritage simple, c'est-à-dire avec une superclasse unique, est autorisé en Java.

Exemple : une sous-classe de `Point`

```
import java.awt.*;
public class DessinPoint extends Point {
  private Color couleur;
  public DessinPoint(int x, int y, Color c) { // attribut supplémentaire
    super(x, y);                              // appel constructeur de Point
    couleur = c;
  }
  public void dessine(Graphics g) {           // méthode supplémentaire
     g.setColor(couleur);
     g.fillRect(getX()-2, getY()-2, 4, 4);
  }
}
```

Ainsi programmée, la sous-classe `DessinPoint` possède trois attributs (dont deux hérités), un constructeur (il n'y a pas héritage des constructeurs), et cinq méthodes (dont quatre héritées).

2.4.2. Upcast et downcast

Le *cast* (ou transtypage) est l'opération de changement du type déclaré d'un objet, c'est-à-dire du type de sa référence. Le type réel d'un objet ne peut jamais être modifié : il reste celui associé à sa création par l'opérateur `new`.

Changer le type déclaré vers une classe moins spécifique (*upcast* ou généralisation) est une opération toujours sûre.

Exemple :

```
DessinPoint dp = new DessinPoint();
Point p = dp;                       // upcast toujours possible
```

Changer le type déclaré vers une classe plus spécifique (*downcast* ou spécialisation) est une opération à risque, comme le montre l'exemple qui suit.

Exemple : (suite de l'exemple précédent)

```
DessinPoint dp1 = (DessinPoint) p; // downcast OK car p est un DessinPoint
// supposons que Point et Rectangle implantent l'interface Forme
Forme f = (Forme) p;               // upcast toujours possible
Rectangle r = (Rectangle) f;       // compile mais lève une exception
// à l'exécution car un DessinPoint n'est pas un Rectangle
```

2.4.3. Redéfinition et surcharge

Une sous-classe peut *redéfinir* une méthode (*override*) en gardant le même profil : nom, types et ordre des paramètres, type retourné. Cette nouvelle implantation remplace celle héritée et peut faire appel ou non au code hérité par `super.methode()`.

Une sous-classe peut aussi *surcharger* une méthode (*overload*) en conservant son nom mais en changeant des éléments de son profil (nombre et position des paramètres, le type retourné n'est pas pris en compte). Il s'agit dans ce cas d'ajouter une variante spécialisée à la méthode héritée.

Exemple : redéfinition de méthode

On veut implanter une liste de points en garantissant qu'elle ne contient jamais `null`.

On peut penser redéfinir la méthode `boolean add(Point p)` de `ArrayList <Point>` en interdisant l'ajout de `null`.

```
import java.util.*;
public class ListePoints extends ArrayList<Point> {
  public boolean add(Point p) {
    if (p != null) return super.add(p);
    else return false;
  }
}
```

C'est tout à fait inefficace, car il est possible de passer par d'autres méthodes de `ArrayList`, comme `addAll()` :

```
List<Point> l = new ArrayList<Point>();
l.add(null);
ListePoints lp = new ListePoints();
lp.addAll(l);
```

Pour garantir l'absence d'élément `null` il faudrait redéfinir *toutes* les autres méthodes susceptibles d'ajouter des éléments. En effet *l'héritage implique toujours la « récupération » de toutes les méthodes par la sous-classe*.

2.4.4. Héritage, couplage et encapsulation

En termes plus généraux, on peut dire que l'héritage *couple* fortement les deux classes. Dans beaucoup de cas, il vaut mieux recourir à la *délégation* (composition) en suivant le conseil célèbre du Gof [Gam+95] : « *favorisez la composition par rapport à l'héritage* ».

Comme expliqué en détail au paragraphe suivant, cela revient à écrire pour l'exemple précédent :

```
import java.util.*;
public class ListePoints {
  private ArrayList<Point> list;
  public ListePoints() {
    list = new ArrayList<Point>();
  }
  public void add(Point p) {
    if (p != null) list.add(p);
  }
}
```

Ici, la seule manière de modifier la liste est de passer par la méthode `add()`, car les autres méthodes ne sont plus disponibles. Le contrôle de l'absence de `null` devient efficace.

On peut affirmer également que *l'héritage brise la notion d'encapsulation*. En effet, la sous-classe peut dépendre des détails d'implantation de la superclasse pour son propre fonctionnement. Si l'implantation de la superclasse évolue, la sous-classe peut éventuellement ne plus fonctionner correctement, même sans que son code ait été modifié !

2.5 DÉLÉGATION

On parle de *délégation*, ou composition, quand une classe (cliente) délègue à une autre classe (serveuse) une partie de son activité. Il y a donc réutilisation de code entre classes. Il y a parfois aussi la volonté de séparer du reste d'une classe ce qui est susceptible d'évoluer fréquemment.

La délégation s'implante grâce à un attribut de la classe cliente contenant une référence vers la classe serveuse. En changeant la référence on peut changer le comportement, y compris pendant l'exécution.

Exemple : translation d'un cercle

La translation du cercle réutilise la translation de son centre.

```
public class Cercle {
  private Point centre; // attribut qui référence une autre classe
  private double rayon;
  public Cercle(Point c, double r) { // constructeur
    centre = c;
    rayon = r;
  }
  public void translater(int dx, int dy) {
    centre.translater(dx,dy);        // réutilisation de code
  }
  public String toString() {
    return centre.getX()+" "+centre.getY()+" "+rayon~;
  }
}
```

À noter au passage qu'avec cette conception, deux cercles peuvent partager le même `Point` centre, ce qui peut poser problème :

```
Point p1 = new Point(10, 20);
Cercle c1 = new Cercle(p1, 5);
Cercle c2 = new Cercle(p1, 8);
System.out.println(c1); System.out.println(c2);
c1.translater(10, 10);
System.out.println(c1); System.out.println(c2);
```

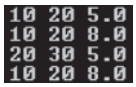
donne :

```
10 20 5.0
10 20 8.0
20 30 5.0
20 30 8.0
```

Les deux cercles ont été translatés par effet de bord ! Il est préférable de rendre le cercle et le centre dépendants, en concevant ainsi le constructeur de `Cercle` :

```
public Cercle(Point c, double r) {
  centre = new Point (c.getX(), c.getY());
  rayon = r;
}
```

Dans cette conception, chaque cercle a son propre `Point` centre. La translation de `c1` est donc sans effet sur `c2` :



2.6 INTERFACE

Une *interface* est la définition abstraite d'un type, indépendamment de la façon dont il est implanté (*type abstrait*). Concrètement, c'est un ensemble de méthodes publiques abstraites et accessoirement de constantes de classe `static final`.

Il est très conseillé de typer aux maximum les variables par des interfaces, qui évoluent peu, plutôt que par des implantations, susceptibles d'évoluer bien plus souvent.

Exemple : une interface

```
public interface Redimensionnable {
  void deformer(double facteur);   // public abstract est implicite
}
// la classe Cercle implante (respecte) l'interface Redimensionnable
public class Cercle implements Redimensionnable {
  public void deformer(double facteur) {
    rayon = rayon * facteur;
    // ...
  }
}
```

Une interface définit un *type statique*. Une référence à une interface peut désigner toute instance de toute classe qui implante cette interface.

Exemple : une référence à une interface

```
Cercle c = new Cercle(new Point(10,10), 5);
Redimensionnable r = c;
```

Les variables `r` et `c` réfèrent le même objet. On peut écrire `r.deformer(2.5)` ou `c.deformer(2.5)`.

En privilégiant les interfaces comme types pour les variables, attributs des classes, paramètres et types de retour des méthodes on rend le code indépendant des implantations et donc plus flexible et plus facilement réutilisable. Le seul endroit où la classe effective est toujours indispensable est l'opérateur `new`.

Quand une interface est disponible, il est toujours préférable de l'utiliser comme type statique. C'est ce qu'exprime le deuxième conseil célèbre du GoF [Gam+95] : « *programmez avec des interfaces, pas des implantations* ».

Les interfaces permettent également une forme *d'héritage multiple restreint*. En effet, une classe peut implanter (`implements`) plusieurs interfaces. À noter qu'une interface peut elle-même hériter (`extends`) de plusieurs interfaces ! C'est le seul cas où l'héritage multiple est autorisé en Java.

2.7 POLYMORPHISME

Le *polymorphisme* est la capacité qu'offre le langage à exécuter une méthode en fonction du type réel de l'objet concerné par l'appel. Le code qui est appelé dépend de la *classe* de l'objet destinataire (son type dynamique ou réel) et non du *type* de la variable dans laquelle il est stocké (son type statique ou déclaré), qui peut être celui d'une de ses superclasses ou d'une des interfaces qu'il implante. Le polymorphisme permet donc d'écrire de manière « *générique* » certaines parties d'un code. Le même code peut donner lieu à des appels de méthodes différents selon les types dynamiques des objets.

Exemple :

```
public interface Parleur {
  void parle();
}
public class Chien implements Parleur {
  public void parle() {
    System.out.println("ouaf ouaf");
  }
}
public class Chat implements Parleur {
  public void parle() {
    System.out.println("miaou miaou");
  }
}
public class Polymorphique {
  // méthode générique écrite indépendamment des types réels (Chat, Chien...)
  private static void parler(Parleur[] tab) {
    for(Parleur p: tab) p.parle();
  }
  public static void main(String[] args) {
    Parleur[] animaux = {new Chien(), new Chat()};
    parler(animaux);
  }
}
```

affiche :

2.8 PAQUET

Un *paquet* (paquetage ou *package*) est un moyen de regrouper des classes, des interfaces, des (sous-)paquets afin de faciliter la modularité.

La notation pointée sert à accéder aux éléments contenus dans un paquet.

Exemple : `java.util.ArrayList` indique que la classe `ArrayList` est dans le sous-paquet `util` du paquet `java`.

La déclaration `package A.B`, qui doit figurer en début de fichier source, indique que les déclarations qui suivent font partie du paquet `A.B`. En son absence, c'est le paquet par défaut qui est utilisé.

La déclaration `import A.B.x` rend visible la classe `x` du paquet `A.B`. On peut l'appeler simplement `x` au lieu de `A.B.x`.

La déclaration `import A.B.*` rend visible toutes les classes du paquet `A.B`. À noter que les classes du sous-paquet `C` de `B`, comme `A.B.C.y`, ne sont pas rendues visibles par cette déclaration. À noter également que le paquet `java.lang`, qui contient les classes de base de Java, est automatiquement importé par le compilateur.

2.9 THREADS

Java supporte les *« processus légers »* ou *threads*. Ceux-ci permettent de gérer des activités parallèles au sein d'un même programme. Le contrôle de leur exécution peut se faire, au moins en partie, au sein du programme. Les threads peuvent communiquer entre eux et se partager des données. Le programme tout entier correspond lui à un *« processus lourd »*, géré par le système d'exploitation.

2.9.1. Création

Lorsqu'une machine virtuelle Java est démarrée, elle crée un premier thread applicatif, le *« thread principal »*, qui appelle la méthode `main` de la classe spécifiée. Ensuite des threads applicatifs peuvent être créés et détruits dynamiquement par le programme. La méthode `System.exit` permet d'arrêter la machine virtuelle. Sinon, l'exécution se poursuit jusqu'à la terminaison de tous les threads applicatifs, y compris le thread principal. En parallèle, un certain nombre de démons gèrent des activités annexes, comme le ramasse-miettes par exemple. Tous les démons sont arrêtés brutalement lorsque le dernier thread applicatif se termine.

Un thread applicatif peut être créé de deux manières. Soit en définissant une classe qui hérite de `Thread` et en redéfinissant la méthode `run` :

```
class X extends Thread {
  public void run() {
    // code de l'activité
  }
  // ...
}
// Utilisation
X x = new X(...);
x.start();
```

Soit en définissant une classe qui implante l'interface `Runnable`, ce qui consiste simplement à implanter la méthode `public void run()`, et en créant une instance de `Thread` avec un objet `Runnable` en paramètre :

```
class X implements Runnable {
  public void run () {
    // code de l'activité
  }
  // ...
}
// utilisation
X x = new X(...);
Thread t = new Thread(x);
t.start();
```

2.9.2. Interruption

Tous les threads applicatifs ainsi créés s'exécutent en simultanéité, au moins apparente. Les machines virtuelles Java récentes savent exploiter les cœurs multiples des processeurs modernes et les possibilités de simultanéité réelle qu'ils offrent.

L'instruction `Thread.sleep(500)` met le thread dans lequel elle se trouve en sommeil pendant au moins 500 millisecondes.

Par ailleurs, un thread `t1` peut en interrompre un autre, `t2`, par `t2.interrupt()`. Ce dernier peut réagir à son interruption, dans la méthode `run()`, par :

```
if (interrupted) {
  //...
}
```

2.9.3. Coordination

Les threads appartiennent à un même programme et peuvent se partager des objets. En général, il faut éviter qu'ils accèdent en même temps au même objet. Ce problème est résolu grâce aux blocs et aux méthodes exclusives (« *synchronisés* »).

Une méthode synchronisée est déclarée avec le mot-clé `synchronized`. Il y a accès exclusif, via un mécanisme de verrou unique, à l'objet auquel la méthode est appliquée. Les threads qui n'ont pas le verrou doivent attendre que le thread qui le possède le rende.

Un bloc synchronisé indique une région critique pour l'accès à un objet : un seul thread peut posséder le verrou d'accès à cet objet.

```
synchronized (unObj) {
  // région critique
}
```

Le programmeur doit veiller à éviter les situations d'*interblocage*, aussi appelées « étreintes mortelles », comme le cas où un thread `t1` possède le verrou d'accès à un objet `o1` et attend celui d'un objet `o2`, alors que le thread `t2` possède le verrou de l'objet `o2` et attend celui de `o1`.

Enfin, il est possible de faire attendre un thread tant qu'un autre effectue un certain travail grâce aux méthodes `wait()`, `notify()` et `notifyAll()`.

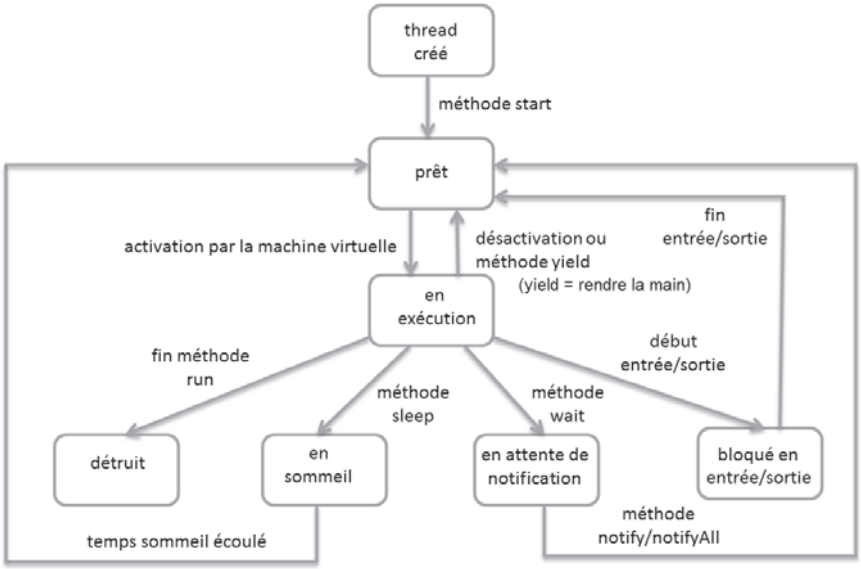


FIGURE 2.4 Les états possibles d'un thread

2.10 COMPOSANTS

Le concept de *composant* doit être clairement distingué du concept de classe, même si la description d'un composant se fait en général à l'aide d'une classe. Un composant ne peut pas vivre en dehors d'un environnement spécial que l'on appelle un « conteneur » (*container*). Un tel conteneur instancie les composants et contrôle leurs évolutions, relie les composants selon leurs interfaces requises et fournies, et procure des services aux composants que le développeur n'a pas à coder (transactions, sécurité, journalisation – *logging*). Contrairement

aux classes, les composants ne sont donc jamais créés par les programmeurs à l'aide de l'opérateur `new`. Ils offrent des services aux autres composants et utilisent des services d'autres composants. Ils sont caractérisés par un « *cycle de vie* », c'est-à-dire un ensemble d'évolutions orchestrées par le conteneur.

2.11 SERVLETS

2.11.1. Définition

Un *servlet* est un composant qui s'exécute dans un conteneur de servlets (comme Tomcat par exemple). Il sert à créer des pages dynamiques « à la volée » et à effectuer des traitements applicatifs côté serveur.

Le scénario de base est le suivant : une requête HTTP, envoyée par un navigateur Web à l'URL associée au servlet, déclenche l'exécution sur le serveur du servlet qui construit et envoie sa réponse (HTML/XML) au client.

La classe `HttpServlet` hérite de `GenericServlet` et fournit une implantation spécifique à HTTP de l'interface générique `Servlet`. Pour l'utiliser, il suffit de surcharger la ou les méthodes `doGet`, `doPost`, `doPut`... correspondant au traitement des requêtes `GET`, `POST`, `PUT`...

Ces méthodes possèdent un paramètre correspondant à la requête HTTP (un objet de la classe `HttpServletRequest`) et un paramètre correspondant à la réponse (un objet de la classe `HttpServletResponse`). À partir de l'objet requête il est possible d'accéder aux paramètres de la requête via la méthode `String getParameter(String nom)`. Ces paramètres sont à vérifier et souvent à convertir. À partir de l'objet réponse il est possible d'accéder au flux sur lequel le servlet va écrire la page HTML de réponse via la méthode `PrintWriter getWriter()`.

Durant le cycle de vie des servlets, le conteneur invoque également les méthodes `init`, à la création du servlet (qui sert en général à initialiser les connexions réseaux et BD, et à récupérer des paramètres d'initialisation et de contexte) et `destroy`, à l'arrêt du conteneur (qui sert en général à fermer les connexions réseaux et BD).

Exemple : le servlet (`Bonjour.java`)

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Bonjour extends HttpServlet {
  private int compte;
  public void init(ServletConfig config) throws ServletException {
    compte = 0;
  }
  public void doPost(HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException {
    res.setContentType("text/html");   // type MIME de la réponse
    PrintWriter out = res.getWriter();
```