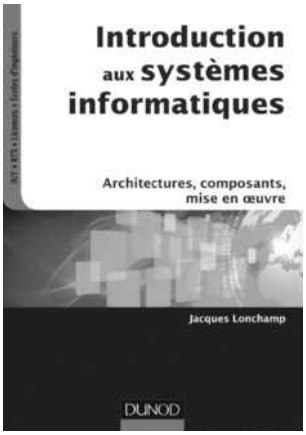
Conception d'applications en Java/JEE

Du même auteur :



Conception d'applications en Java/JEE

Jacques Longchamp
Professeur à l'université de Lorraine

2e édition

DUNOD

Illustration de couverture : Anton Darius. The sollers

© Dunod, 2014, 2019
11 rue Paul Bert, 92240 Malakoff
www.dunod.com
ISBN 978-2-10-079045-6

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* ». Le premier thème est traité dans un autre volume du même auteur [Lon15].

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. 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.

LES NOUVEAUTÉS DE LA DEUXIÈME ÉDITION

Certaines évolutions des langages de programmation peuvent avoir un impact sur la mise en œuvre des schémas de conception et même en créer de nouveaux. C'est le cas de plusieurs évolutions récentes du langage Java, dans ses versions 8 à 11.

– L'apparition des capacités de programmation fonctionnelle en Java 8, à travers les interfaces fonctionnelles et les lambda expressions, modifie la manière d'implanter plusieurs patrons de conception du « GoF ».

– Le patron *map/filter/reduce* simplifie beaucoup la programmation des traitements qui opèrent sur des collections d'objets. Cette approche, typique de la programmation fonctionnelle, est de nature déclarative : la manipulation des collections est décrite comme une simple composition de fonctions. Ce schéma de conception est pris en compte dans cette deuxième édition, car le concept de *stream* en Java 8, en complément des lambda expressions, permet de l'implanter très simplement, y compris de manière parallèle sur les machines multicœurs.

– Enfin, les modules, apparus avec Java 9, constituent une évolution fondamentale dans la manière de structurer les applications Java.

Les principales modifications de cette deuxième édition sont donc les suivantes :

– Le chapitre 2, consacré aux rappels sur Java et JEE, est augmenté d'un paragraphe décrivant les principales nouveautés de Java 8, les lambda expressions et les *streams* et d'un paragraphe décrivant les modules de Java 9. À noter que les versions 10 et 11, séparées des précédentes de seulement six mois, n'apportent pas de nouveautés de la même importance.

– Aux chapitres 4, 5 et 6, des implantations de certains patrons du GoF mettant en œuvre les lambda expressions complètent le texte initial. Le lecteur peut ainsi comparer les anciennes et les nouvelles implantations et mesurer concrètement l'intérêt de l'approche fonctionnelle en Java. Au chapitre 6, une nouvelle implantation du patron Observateur est proposée pour tenir compte de l'obsolescence de classes et interface *Observable* et *Observer* depuis Java 9.

– Le chapitre 12, consacré aux architectures de type flot de données, est complété par la description du schéma *map/filter/reduce* et de sa mise en œuvre avec les *streams* Java.

– L'étude de cas JSE du chapitre 16, est complétée pour illustrer la mise en œuvre du concept de module.

– Du côté de JEE, les principales évolutions apparues avec la version 8, comme le passage à HTML/2 pour les servlets, sont simplement citées, car elles impactent plus la mise en œuvre technique des applications que leur conception.

Enfin, cette seconde édition met plus en valeur certains thèmes devenus majeurs, avec des paragraphes additionnels sur les services Web REST (page 51) et les architectures orientées microservices (page 193).

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. Un ouvrage complémentaire de celui-ci détaille le domaine de l'analyse, ses principes, techniques, notations et démarches [Lon15].

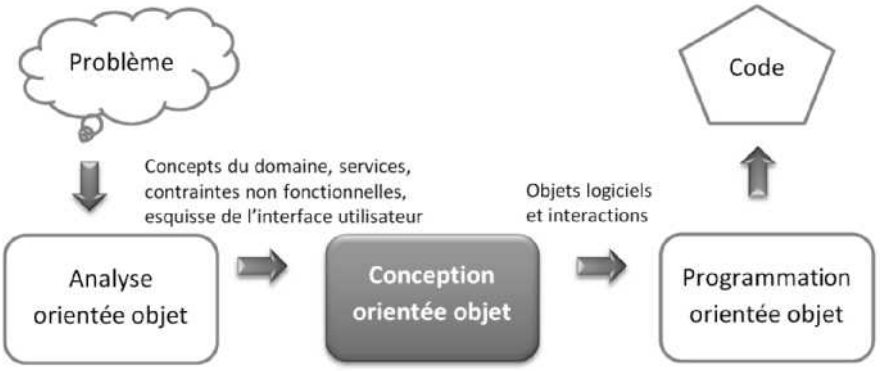


FIGURE 1.1 Place de la conception orientée objet

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 une succession de phases très courtes (quelques semaines au plus), planifiées de manière souple. Ces phases correspondent à la fois à des remaniements et à des ajouts. De telles 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

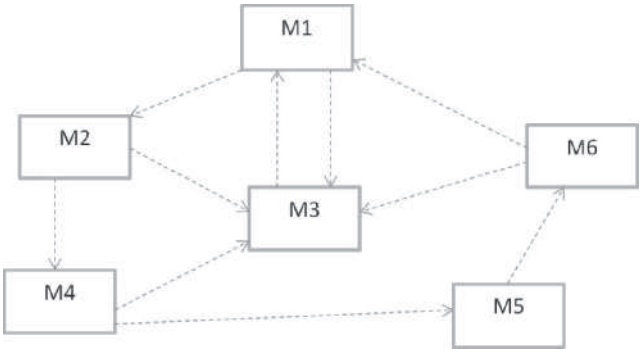
Le domaine de la conception orientée objet est très vaste. Cet ouvrage en fait une présentation synthétique qui préserve cependant la multiplicité des points de vue évoquée ci-dessus : principes théoriques, patrons de conception concrets, styles et patrons architecturaux.

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

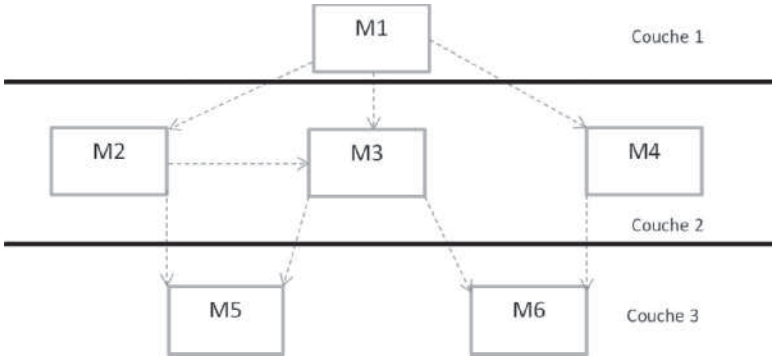
générale ». Au contraire, en raison de leur importance actuelle, un éclairage tout particulier est mis sur les architectures Web, trois-tiers et multi-tiers, dans le contexte JEE. Les patrons spécifiques à l'utilisation de cette plateforme sont explicités. Des illustrations concrètes de mise en œuvre 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