

Référence

Les design patterns en Java

Les 23 modèles
de conception
fondamentaux

Steven John Metsker

William C. Wake

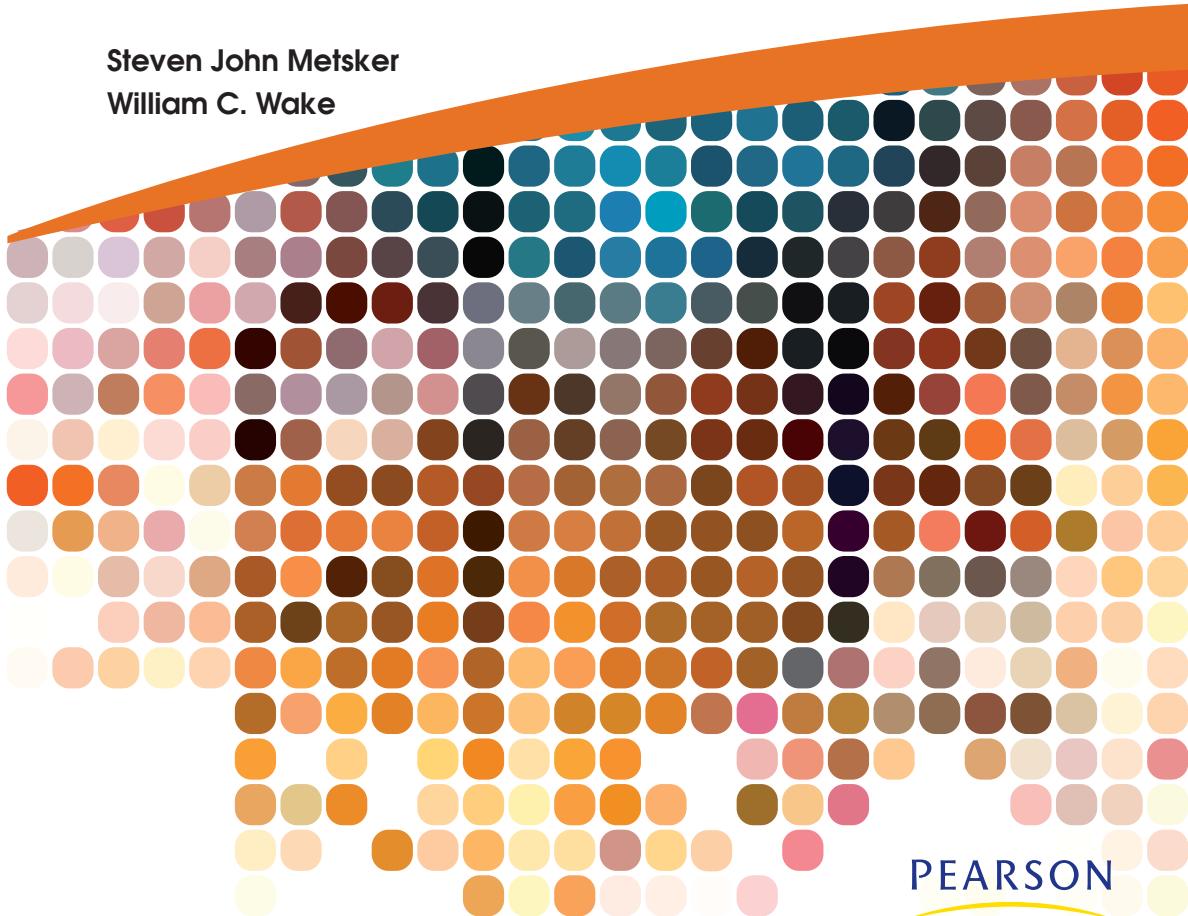
Réseaux
et télécom

Programmation

Génie logiciel

Sécurité

Système
d'exploitation



PEARSON

Les Design Patterns en Java

Les 23 modèles de conception fondamentaux

**Steven John Metsker
et William C. Wake**



Pearson Education France a apporté le plus grand soin à la réalisation de ce livre afin de vous fournir une information complète et fiable. Cependant, Pearson Education France n'assume de responsabilités, ni pour son utilisation, ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

Les exemples ou les programmes présents dans cet ouvrage sont fournis pour illustrer les descriptions théoriques. Ils ne sont en aucun cas destinés à une utilisation commerciale ou professionnelle.

Pearson Education France ne pourra en aucun cas être tenu pour responsable des préjudices ou dommages de quelque nature que ce soit pouvant résulter de l'utilisation de ces exemples ou programmes.

Tous les noms de produits ou marques cités dans ce livre sont des marques déposées par leurs propriétaires respectifs.

Publié par Pearson Education France
47 bis, rue des Vinaigriers
75010 PARIS
Tél. : 01 72 74 90 00
www.pearson.fr

Mise en pages : TyPAO

ISBN : 978-2-7440-4097-9
Copyright © 2009 Pearson Education France
Tous droits réservés

Titre original : *Design Patterns in Java*

Traduit de l'américain par Freenet Sofor ltd

ISBN original : 0-321-33302-0
Copyright © 2006 by Addison-Wesley
Tous droits réservés

Aucune représentation ou reproduction, même partielle, autre que celles prévues à l'article L. 122-5 2° et 3° a) du code de la propriété intellectuelle ne peut être faite sans l'autorisation expresse de Pearson Education France ou, le cas échéant, sans le respect des modalités prévues à l'article L. 122-10 dudit code.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Table des matières

Préface	1
Conventions de codage	1
Remerciements	2
 Chapitre 1. Introduction	3
Qu'est-ce qu'un pattern ?	3
Qu'est-ce qu'un pattern de conception ?	4
Liste des patterns décrits dans l'ouvrage	5
Java	7
UML	7
Exercices	8
Organisation du livre	9
Oozinoz	10
Résumé	11

Partie I

Patterns d'interface

 Chapitre 2. Introduction aux interfaces	15
Interfaces et classes abstraites	16
Interfaces et obligations	17
Résumé	19
Au-delà des interfaces ordinaires	19
 Chapitre 3. ADAPTER	21
Adaptation à une interface	21
Adaptateurs de classe et d'objet	25

Adaptation de données pour un widget <code>JTable</code>	29
Identification d'adaptateurs	33
Résumé	34
Chapitre 4. FACADE	35
Façades, utilitaires et démos	36
Refactorisation pour appliquer FACADE	37
Résumé	46
Chapitre 5. COMPOSITE	47
Un composite ordinaire	47
Comportement récursif dans les objets composites	48
Objets composites, arbres et cycles	50
Des composites avec des cycles	55
Conséquences des cycles	59
Résumé	60
Chapitre 6. BRIDGE	61
Une abstraction ordinaire	61
De l'abstraction au pattern BRIDGE	64
Des drivers en tant que BRIDGE	66
Drivers de base de données	67
Résumé	69

Partie II

Patterns de responsabilité

Chapitre 7. Introduction à la responsabilité	73
Responsabilité ordinaire	73
Contrôle de la responsabilité grâce à la visibilité	75
Résumé	77
Au-delà de la responsabilité ordinaire	77

Chapitre 8. SINGLETON	79
Le mécanisme de SINGLETON	79
Singletons et threads	81
Identification de singltons	82
Résumé	84
Chapitre 9. OBSERVER	85
Un exemple classique : OBSERVER dans les interfaces utilisateurs	85
Modèle-Vue-Contrôleur	90
Maintenance d'un objet Observable	96
Résumé	99
Chapitre 10. MEDIATOR	101
Un exemple classique : médiateur de GUI	101
Médiateur d'intégrité relationnelle	106
Résumé	112
Chapitre 11. PROXY	115
Un exemple classique : proxy d'image	115
Reconsidération des proxies d'image	120
Proxy distant	122
Proxy dynamique	128
Résumé	133
Chapitre 12. CHAIN OF RESPONSABILITY	135
Une chaîne de responsabilités ordinaire	135
Refactorisation pour appliquer CHAIN OF RESPONSABILITY	137
Ancrage d'une chaîne de responsabilités	140
CHAIN OF RESPONSABILITY sans COMPOSITE	142
Résumé	142
Chapitre 13. FLYWEIGHT	143
Immuabilité	143
Extraction de la partie immuable d'un flyweight	144
Partage des objets flyweight	146
Résumé	149

Partie III**Patterns de construction**

Chapitre 14. Introduction à la construction	153
Quelques défis de construction	153
Résumé	155
Au-delà de la construction ordinaire	155
Chapitre 15. BUILDER	157
Un objet constructeur ordinaire	157
Construction avec des contraintes	160
Un builder tolérant	163
Résumé	164
Chapitre 16. FACTORY METHOD	165
Un exemple classique : des itérateurs	165
Identification de FACTORY METHOD	166
Garder le contrôle sur le choix de la classe à instancier	167
Application de FACTORY METHOD dans une hiérarchie parallèle	169
Résumé	171
Chapitre 17. ABSTRACT FACTORY	173
Un exemple classique : le kit de GUI	173
Classe FACTORY abstraite et pattern FACTORY METHOD	178
Packages et classes factory abstraites	182
Résumé	182
Chapitre 18. PROTOTYPE	183
Des prototypes en tant qu'objets factory	183
Prototypage avec des clones	185
Résumé	187
Chapitre 19. MEMENTO	189
Un exemple classique : défaire une opération	189
Durée de vie des mémentos	196

Persistance des mémentos entre les sessions	197
Résumé	200

Partie IV

Patterns d'opération

Chapitre 20. Introduction aux opérations	203
Opérations et méthodes	203
Signatures	205
Exceptions	205
Algorithmes et polymorphisme	206
Résumé	208
Au-delà des opérations ordinaires	209
 Chapitre 21. TEMPLATE METHOD	211
Un exemple classique : algorithme de tri	211
Complétion d'un algorithme	215
Hooks	218
Refactorisation pour appliquer TEMPLATE METHOD	219
Résumé	221
 Chapitre 22. STATE	223
Modélisation d'états	223
Refactorisation pour appliquer STATE	227
Etats constants	231
Résumé	233
 Chapitre 23. STRATEGY	235
Modélisation de stratégies	236
Refactorisation pour appliquer STRATEGY	238
Comparaison de STRATEGY et STATE	242
Comparaison de STRATEGY et TEMPLATE METHOD	243
Résumé	243

Chapitre 24. COMMAND	245
Un exemple classique : commandes de menus	245
Emploi de COMMAND pour fournir un service	248
Hooks	249
COMMAND en relation avec d'autres patterns	251
Résumé	252
Chapitre 25. INTERPRETER	253
Un exemple de INTERPRETER	254
Interpréteurs, langages et analyseurs syntaxiques	265
Résumé	266

Partie V

Patterns d'extension

Chapitre 26. Introduction aux extensions	269
Principes de la conception orientée objet	269
Le principe de substitution de Liskov	270
La loi de Demeter	271
Elimination des erreurs potentielles	273
Au-delà des extensions ordinaires	273
Résumé	274
Chapitre 27. DECORATOR	277
Un exemple classique : flux d'E/S et objets Writer	277
Enveloppeurs de fonctions	285
DECORATOR en relation avec d'autres patterns	292
Résumé	293
Chapitre 28. ITERATOR	295
Itération ordinaire	295
Itération avec sécurité inter-threads	297
Itération sur un objet composite	303
Ajout d'un niveau de profondeur à un énumérateur	310
Enumération des feuilles	311
Résumé	313

Chapitre 29. VISITOR	315
Application de VISITOR	315
Un VISITOR ordinaire	318
Cycles et VISITOR	323
Risques de VISITOR	328
Résumé	330

Partie VI

Annexes

Annexe A. Recommandations	333
Tirer le meilleur parti du livre	333
Connaître ses classiques	334
Appliquer les patterns	334
Continuer d'apprendre	336
Annexe B. Solutions	337
Introduction aux interfaces	337
Solution 2.1	337
Solution 2.2	338
Solution 2.3	338
ADAPTER	338
Solution 3.1	338
Solution 3.2	339
Solution 3.3	340
Solution 3.4	341
Solution 3.5	341
Solution 3.6	342
FACADE	342
Solution 4.1	342
Solution 4.2	343
Solution 4.3	343
Solution 4.4	344

COMPOSITE	345
Solution 5.1	345
Solution 5.2	346
Solution 5.3	346
Solution 5.4	347
Solution 5.5	347
Solution 5.6	348
BRIDGE	348
Solution 6.1	348
Solution 6.2	348
Solution 6.3	349
Solution 6.4	349
Solution 6.5	350
Introduction à la responsabilité	350
Solution 7.1	350
Solution 7.2	351
Solution 7.3	352
Solution 7.4	353
SINGLETON	353
Solution 8.1	353
Solution 8.2	353
Solution 8.3	353
Solution 8.4	354
OBSERVER	354
Solution 9.1	354
Solution 9.2	355
Solution 9.3	356
Solution 9.4	356
Solution 9.5	357
Solution 9.6	357
Solution 9.7	358
MEDIATOR	359
Solution 10.1	359
Solution 10.2	360
Solution 10.3	361
Solution 10.4	361
Solution 10.5	362

PROXY	362
Solution 11.1	362
Solution 11.2	363
Solution 11.3	363
Solution 11.4	363
Solution 11.5	364
CHAIN OF RESPONSABILITY	364
Solution 12.1	364
Solution 12.2	365
Solution 12.3	366
Solution 12.4	366
Solution 12.5	367
FLYWEIGHT	368
Solution 13.1	368
Solution 13.2	369
Solution 13.3	370
Solution 13.4	370
Introduction à la construction	371
Solution 14.1	371
Solution 14.2	372
Solution 14.3	372
BUILDER	373
Solution 15.1	373
Solution 15.2	373
Solution 15.3	374
Solution 15.4	374
FACTORY METHOD	375
Solution 16.1	375
Solution 16.2	376
Solution 16.3	376
Solution 16.4	376
Solution 16.5	377
Solution 16.6	378
Solution 16.7	378
ABSTRACT FACTORY	379
Solution 17.1	379
Solution 17.2	380

Solution 17.3	380
Solution 17.4	381
Solution 17.5	381
PROTOTYPE	382
Solution 18.1	382
Solution 18.2	383
Solution 18.3	383
Solution 18.4	384
MEMENTO	384
Solution 19.1	384
Solution 19.2	385
Solution 19.3	385
Solution 19.4	386
Solution 19.5	386
Introduction aux opérations	387
Solution 20.1	387
Solution 20.2	387
Solution 20.3	388
Solution 20.4	388
Solution 20.5	388
TEMPLATE METHOD	389
Solution 21.1	389
Solution 21.2	389
Solution 21.3	390
Solution 21.4	390
STATE	390
Solution 22.1	390
Solution 22.2	390
Solution 22.3	391
Solution 22.4	391
STRATEGY	392
Solution 23.1	392
Solution 23.2	392
Solution 23.3	392
Solution 23.4	393

COMMAND	393
Solution 24.1	393
Solution 24.2	393
Solution 24.3	395
Solution 24.4	395
Solution 24.5	396
Solution 24.6	396
INTERPRETER	396
Solution 25.1 396	
Solution 25.2	397
Solution 25.3	397
Solution 25.4	397
Introduction aux extensions	398
Solution 26.1 398	
Solution 26.2	398
Solution 26.3	398
Solution 26.4	399
DECORATOR	399
Solution 27.1 399	
Solution 27.2	400
Solution 27.3	401
Solution 27.4	401
ITERATOR	401
Solution 28.1 401	
Solution 28.2	402
Solution 28.3	402
Solution 28.4	402
VISITOR	403
Solution 29.1 403	
Solution 29.2	403
Solution 29.3	403
Solution 29.4	404
Solution 29.5	404
Annexe C. Code source d’Oozinoz	405
Obtention et utilisation du code source	405
Construction du code d’Oozinoz	406

Test du code avec JUnit	406
Localiser les fichiers	406
Résumé	407
Annexe D. Introduction à UML	409
Classes	409
Relations entre classes	412
Interfaces	414
Objets	414
Etats	416
Glossaire	417
Bibliographie	425
Index	427

Préface

Les patterns de conception sont des solutions de niveaux classe et méthode à des problèmes courants dans le développement orienté objet. Si vous êtes déjà un programmeur Java intermédiaire et souhaitez devenir avancé, ou bien si vous êtes avancé mais n'avez pas encore étudié les patterns de conception, ce livre est pour vous.

Il adopte une approche de cahier d'exercices, chaque chapitre étant consacré à un pattern particulier. En plus d'expliquer le pattern en question, chaque chapitre inclut un certain nombre d'exercices vous demandant d'expliquer quelque chose ou de développer du code pour résoudre un problème.

Nous vous recommandons vivement de prendre le temps d'effectuer chaque exercice lorsque vous tombez dessus plutôt que de lire le livre d'une traite. En mettant en pratique vos connaissances au fur et à mesure de leur acquisition, vous apprenez mieux, même si vous ne faites pas plus d'un ou deux chapitres par semaine.

Conventions de codage

Le code des exemples présentés dans ce livre est disponible en ligne. Voyez l'Annexe C pour savoir comment l'obtenir.

Nous avons utilisé le plus souvent un style cohérent avec les conventions de codage de Sun. Les accolades ont été omises lorsque c'était possible. Nous avons dû faire quelques compromis pour nous adapter au format du livre. Pour respecter les colonnes étroites, les noms de variables sont parfois plus courts que ceux que nous employons habituellement. Et pour éviter les complications du contrôle de code source, nous avons distingué les multiples versions d'un même fichier en accolant un chiffre à son nom (par exemple, ShowBallistics2). Vous devriez normalement utiliser le contrôle de code source et travailler seulement avec la dernière version d'une classe.

Remerciements

Nous tenons à remercier le défunt John Vlissides pour ses encouragements et ses recommandations concernant ce livre et d'autres. John, éditeur de la collection Software Patterns Series et coauteur de l'ouvrage original *Design Patterns*, était pour nous un ami et une inspiration.

En plus de nous appuyer largement sur *Design Patterns*, nous nous sommes aussi inspirés de nombreux autres livres. Voyez pour cela la bibliographie en fin d'ouvrage. En particulier, *The Unified Modeling Language User Guide (le Guide de l'utilisateur UML)* [Booch, Rumbaugh, et Jacobsen 1999] donne une explication claire d'UML, et *Java™ in a Nutshell (Java en concentré : Manuel de référence pour Java)* [Flanagan 2005] constitue une aide concise et précise sur Java. *The Chemistry of Fireworks* [Russell 2000] nous a servi de source d'informations pour élaborer nos exemples pyrotechniques réalistes.

Enfin, nous sommes reconnaissants à toute l'équipe de production pour son travail acharné et son dévouement.

Steve Metsker (Steve.Metsker@acm.org)

Bill Wake (William.Wake@acm.org)

1

Introduction

Ce livre couvre le même ensemble de techniques que l'ouvrage de référence *Design Patterns*, d'Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides [Gamma et al. 1995], et propose des exemples en Java. Il inclut de nombreux exercices conçus pour vous aider à développer votre aptitude à appliquer les patterns de conception dans vos programmes.

Il s'adresse aux développeurs qui connaissent Java et souhaitent améliorer leurs compétences en tant que concepteurs.

Qu'est-ce qu'un pattern ?

Un **pattern**, ou modèle, est un moyen d'accomplir quelque chose, un moyen d'atteindre un objectif, une technique. Le principe est de compiler les méthodes éprouvées qui s'appliquent à de nombreux types d'efforts, tels que la fabrication d'aliments, d'artifices, de logiciels, ou autres. Dans n'importe quel art ou métier nouveau en voie de maturation, ses pratiquants commencent, à un moment donné, à élaborer des méthodes communes efficaces pour parvenir à leurs buts et résoudre des problèmes dans différents contextes. Cette communauté invente aussi généralement un jargon pour pouvoir discuter de son savoir-faire. Une partie de cette terminologie a trait aux modèles, ou techniques établies, permettant d'obtenir certains résultats. A mesure que cet art se développe et que son jargon s'étoffe, les auteurs commencent à jouer un rôle important. En documentant les modèles de cet art, ils contribuent à standardiser son jargon et à faire connaître ses techniques.

Christopher Alexander a été un des premiers auteurs à compiler les meilleures pratiques d'un métier en documentant ses modèles. Son travail concerne l'architecture, celle des immeubles et non des logiciels. Dans *A Pattern Language: Towns, Buildings Construction* (Alexander, Ishikouwa, et Silverstein 1977), il décrit des modèles permettant de bâtir avec succès des immeubles et des villes. Cet ouvrage est puissant et a influencé la communauté logicielle notamment en raison du sens qu'il donne au terme *objectif (intent)*.

Vous pourriez penser que les modèles architecturaux servent principalement à concevoir des immeubles. En fait, Alexander établit clairement que leur objectif est de servir et d'inspirer les gens qui occuperont les immeubles et les villes conçus d'après ces modèles. Son travail a montré que les modèles sont un excellent moyen de saisir et de transmettre le savoir-faire et la sagesse d'un art. Il précise également que comprendre et documenter correctement cet objectif est essentiel, philosophique et difficile.

La communauté informatique a fait sienne cette approche en créant de nombreux ouvrages qui documentent des modèles de développement logiciel. Ces livres consignent les meilleures pratiques en matière de processus logiciels, d'analyse logicielle, d'architecture de haut niveau, et de conception de niveau classe. Il en apparaît de nouveaux chaque année. Lisez les critiques et les commentaires de lecteurs pour faire un choix judicieux.

Qu'est-ce qu'un pattern de conception ?

Un **pattern de conception** (*design pattern*) est un modèle qui utilise des classes et leurs méthodes dans un langage orienté objet. Les développeurs commencent souvent à s'intéresser à la conception seulement lorsqu'ils maîtrisent un langage de programmation et écrivent du code depuis longtemps. Il vous est probablement déjà arrivé de remarquer que du code écrit par quelqu'un d'autre semblait plus simple et plus efficace que le vôtre, auquel cas vous avez dû vous demander comment son développeur était parvenu à une telle simplicité. Les patterns de conception interviennent un niveau au-dessus du code et indiquent typiquement comment atteindre un but en n'utilisant que quelques classes. Un pattern représente une idée, et non une implémentation particulière.

D'autres développeurs ont découvert avant vous comment programmer efficacement dans les langages orientés objet. Si vous souhaitez devenir un programmeur

Java avancé, vous devriez étudier les patterns de conception, surtout ceux de ce livre – les mêmes que ceux expliqués dans *Design Patterns*.

L'ouvrage *Design Patterns* décrit vingt-trois patterns de conception (pour plus de détails, voir section suivante). De nombreux autres livres ont suivi sur le sujet, aussi dénombre-t-on au moins cent patterns qui valent la peine d'être connus. Les vingt-trois patterns recensés par Gamma, Helm, Johnson et Vlissides ne sont pas forcément les plus importants, mais ils sont néanmoins proches du haut de la liste. Ces auteurs ont donc bien choisi et les patterns qu'ils documentent valent certainement la peine que vous les appreniez. Ils vous serviront de référence lorsque commencerez à étudier les patterns exposés par d'autres sources.

Liste des patterns décrits dans l'ouvrage

Patterns d'interface

ADAPTER (17) fournit l'interface qu'un client attend en utilisant les services d'une classe dont l'interface est différente.

FACADE (33) fournit une interface simplifiant l'emploi d'un sous-système.

COMPOSITE (47) permet aux clients de traiter de façon uniforme des objets individuels et des compositions d'objets.

BRIDGE (63) découpe une classe qui s'appuie sur des opérations abstraites de l'implémentation de ces opérations, permettant ainsi à la classe et à son implémentation de varier indépendamment.

Patterns de responsabilité

SINGLETON (81) garantit qu'une classe ne possède qu'une seule instance, et fournit un point d'accès global à celle-ci.

OBSERVER (87) définit une dépendance du type un-à-plusieurs (1,n) entre des objets de manière à ce que lorsqu'un objet change d'état, tous les objets dépendants en soient notifiés et soient actualisés afin de pouvoir réagir conformément.

MEDIATOR (103) définit un objet qui encapsule la façon dont un ensemble d'objets interagissent. Cela promeut un couplage lâche, évitant aux objets d'avoir à se référer explicitement les uns aux autres, et permet de varier leur interaction indépendamment.

PROXY (117) contrôle l'accès à un objet en fournissant un intermédiaire pour cet objet.

CHAIN OF RESPONSABILITY (137) évite de coupler l'émetteur d'une requête à son récepteur en permettant à plus d'un objet d'y répondre.

FLYWEIGHT (145) utilise le partage pour supporter efficacement un grand nombre d'objets à forte granularité.

Patterns de construction

BUILDER (159) déplace la logique de construction d'un objet en-dehors de la classe à instancier, typiquement pour permettre une construction partielle ou pour simplifier l'objet.

FACTORY METHOD (167) laisse un autre développeur définir l'interface permettant de créer un objet, tout en gardant un contrôle sur le choix de la classe à instancier.

ABSTRACT FACTORY (175) permet la création de familles d'objets ayant un lien ou interdépendants.

PROTOTYPE (187) fournit de nouveaux objets par la copie d'un exemple.

MEMENTO (193) permet le stockage et la restauration de l'état d'un objet.

Patterns d'opération

TEMPLATE METHOD (217) implémente un algorithme dans une méthode, laissant à d'autres classes le soin de définir certaines étapes de l'algorithme.

STATE (229) distribue la logique dépendant de l'état d'un objet à travers plusieurs classes qui représentent chacune un état différent.

STRATEGY (241) encapsule des approches, ou stratégies, alternatives dans des classes distinctes qui implémentent chacune une opération commune.

COMMAND (251) encapsule une requête en tant qu'objet, de manière à pouvoir paramétriser des clients au moyen de divers types de requêtes (de file d'attente, de temps ou de journalisation) et de permettre à un client de préparer un contexte spécial dans lequel émettre la requête.

INTERPRETER (261) permet de composer des objets exécutables d'après un ensemble de règles de composition que vous définissez.

Patterns d'extension

DECORATOR (287) permet de composer dynamiquement le comportement d'un objet.

ITERATOR (305) fournit un moyen d'accéder de façon séquentielle aux éléments d'une collection.

VISITOR (325) permet de définir une nouvelle opération pour une hiérarchie sans changer ses classes.

Java

Les exemples de ce livre utilisent Java, le langage orienté objet (OO) développé par Sun. Ce langage, ses bibliothèques et ses outils associés forment une suite de produits pour le développement et la gestion de systèmes aux architectures mult-niveaux et orientées objet.

L'importance de Java tient en partie au fait qu'il s'agit d'un **langage de consolidation**, c'est-à-dire conçu pour intégrer les points forts des langages précédents. Cette consolidation est la cause de son succès et garantit que les langages futurs tendront à s'inscrire dans sa continuité au lieu de s'en éloigner radicalement. Votre investissement dans Java ne perdra assurément pas de sa valeur, quel que soit le langage qui lui succède.

Les patterns de *Design Patterns* s'appliquent à Java, car, comme Smalltalk, C++ et C#, ils se fondent sur un paradigme classe/instance. Java ressemble beaucoup plus à Smalltalk et à C++ qu'à Prolog ou Self par exemple. Même s'il ne faut pas négliger l'importance de paradigmes concurrents, le paradigme classe/instance constitue une avancée concrète en informatique appliquée. Le présent livre emploie Java en raison de sa popularité et parce que son évolution suit le chemin des langages que nous utiliserons dans les années à venir.

UML

Lorsque les solutions des exercices contiennent du code, ce livre utilise Java. Mais nombre d'exercices vous demandent de dessiner un diagramme illustrant les relations entre des classes, des packages et d'autres éléments. Vous pouvez choisir la notation que vous préférez, mais sachez que ce livre utilise la notation **UML** (*Unified Modeling Language*). Même si vous la connaissez déjà, il peut être utile

d'avoir une référence à portée de main. Vous pouvez consulter deux ouvrages de qualité : *The Unified Modeling Language User Guide (le Guide de l'utilisateur UML)* [Booch, Rumbaugh, et Jacobson 1999] et *UML Distilled* [Fowler et Scott 2003]. Les connaissances minimales dont vous avez besoin pour ce livre sont données dans l'Annexe D consacrée à UML.

Exercices

Même si vous lisez de nombreux ouvrages sur un sujet, vous n'aurez le sentiment de le maîtriser vraiment qu'en le pratiquant. Tant que vous n'appliquerez pas concrètement les connaissances acquises, certaines subtilités et approches alternatives vous échapperont. Le seul moyen de gagner de l'assurance avec les patterns de conception est de les appliquer dans le cadre d'exercices pratiques.

Le problème lorsque l'on apprend en faisant est que l'on peut causer des dégâts. Vous ne pouvez pas appliquer les patterns de conception dans du code en production si vous ne les maîtrisez pas. Mais il faut bien que vous commenciez à les appliquer pour acquérir ce savoir-faire. La solution est de vous familiariser avec les patterns au travers d'exemples de problèmes, où vos erreurs seront sans conséquence mais instructives.

Chaque chapitre de ce livre débute par une courte introduction puis présente progressivement une série d'exercices. Lorsque vous avez trouvé une solution, vous pouvez la comparer aux réponses proposées dans l'Annexe B. Il se peut que la solution du livre adopte une approche différente de la vôtre, vous faisant voir les choses sous une autre perspective.

Vous ne pouvez probablement pas prévoir le temps qu'il vous faudra pour trouver les réponses aux exercices. Si vous consultez d'autres livres, travaillez avec un collègue et écrivez des échantillons de code pour vérifier votre solution, c'est parfait ! Vous ne regretterez pas l'énergie et le temps investis.

Un avertissement : si vous vous contentez de lire les solutions immédiatement après avoir lu un exercice, vous ne tirerez pas un grand enseignement de ce livre. Ces solutions ne vous seront daucune utilité si vous n'élaborez pas d'abord les vôtres pour pouvoir ensuite les leur comparer et tirer les leçons de vos erreurs.

Organisation du livre

Il existe de nombreuses façons d'organiser et de classer les patterns de conception. Vous pourriez les organiser en fonction de leurs similitudes sur le plan structurel, ou bien suivre l'ordre de *Design Patterns*. Mais l'aspect le plus important d'un pattern est son objectif, c'est-à-dire la valeur potentielle liée à son application. Le présent livre organise les vingt-trois patterns de *Design Patterns* en fonction de leur objectif.

Reste ensuite à déterminer comment catégoriser ces objectifs. Nous sommes partis du principe que l'objectif d'un pattern de conception peut généralement être exprimé comme étant le besoin d'aller plus loin que les fonctionnalités ordinaires intégrées à Java. Par exemple, Java offre un large support pour la définition des interfaces implémentées par les classes. Mais si vous disposez déjà d'une classe dont vous aimeriez modifier l'interface pour qu'elle corresponde aux exigences d'un client, vous pourriez décider d'appliquer le pattern ADAPTER. L'objectif de ce pattern est de vous aider à complémenter les fonctionnalités d'interface intégrées à Java.

Ce livre regroupe les patterns de conception en cinq catégories que voici :

1. Interfaces.
2. Responsabilité.
3. Construction.
4. Opérations.
5. Extensions.

Ces cinq catégories correspondent aux cinq parties du livre. Chaque partie débute par un chapitre qui présente et remet en question les fonctionnalités Java liées au type d'objectif dont il est question. Par exemple, le premier chapitre de la Partie I traite des interfaces Java ordinaires. Il vous amène à réfléchir sur la structure des interfaces Java, notamment en les comparant aux classes abstraites. Les autres chapitres de cette partie décrivent les patterns qui ont pour principal objectif de définir une interface, c'est-à-dire l'ensemble des méthodes qu'un client peut appeler à partir d'un fournisseur de services. Chacun d'eux répond à un besoin que les interfaces Java ne peuvent satisfaire à elles seules.

Ce classement des patterns par objectifs ne signifie pas que chaque pattern supporte seulement un type d'objectif. Lorsqu'il en supporte plusieurs, il fait l'objet d'un chapitre entier dans la première partie à laquelle il s'applique puis il est mentionné brièvement dans les autres parties concernées. Le Tableau 1.1 illustre la catégorisation sous-jacente à l'organisation du livre.

Tableau 1.1 : Une catégorisation des patterns par objectifs

<i>Objectif</i>	<i>Patterns</i>
Interfaces	ADAPTER, FACADE, COMPOSITE, BRIDGE
Responsabilité	SINGLETON, OBSERVER, MEDIATOR, PROXY, CHAIN OF RESPONSIBILITY, FLYWEIGHT
Construction	BUILDER, FACTORY METHOD, ABSTRACT FACTORY, PROTOTYPE, MEMENTO
Opérations	TEMPLATE METHOD, STATE, STRATEGY, COMMAND, INTERPRETER
Extensions	DECORATOR, ITERATOR, VISITOR

Nous espérons que ce classement vous amènera à vous interroger. Pensez-vous aussi que SINGLETON a trait à la responsabilité, et non à la construction ? COMPOSITE est-il réellement un pattern d'interface ? Toute catégorisation est subjective. Mais vous conviendrez certainement que le fait de réfléchir à l'objectif des patterns et à la façon de les appliquer est un exercice très utile.

Oozinoz

Les exercices de ce livre citent tous des exemples d'Oozinoz Fireworks, une entreprise fictive qui fabrique et vend des pièces pour feux d'artifice et organise des événements pyrotechniques. Vous pouvez vous procurer le code de ces exemples à l'adresse www.oozinoz.com. Pour en savoir plus sur la compilation et le test du code, voyez l'Annexe C.

Résumé

Les patterns de conception distillent une sagesse vieille de quelques dizaines d'années qui établit un jargon standard, permettant aux développeurs de nommer les concepts qu'ils appliquent. Ceux abordés dans l'ouvrage de référence *Design Patterns* font partie des patterns de niveau classe les plus utiles et méritent que vous les appreniez. Le présent livre reprend ces patterns mais utilise Java et ses bibliothèques pour ses exemples et exercices. En réalisant les exercices proposés, vous apprendrez à reconnaître et à appliquer une part importante de la sagesse de la communauté logicielle.

I

Patterns d'interface

Introduction aux interfaces

Pour parler de manière abstraite, l'**interface** d'une classe est l'ensemble des méthodes et champs de la classe auxquels des objets d'autres classes sont autorisés à accéder. Elle constitue généralement un engagement que les méthodes accompliront l'opération signifiée par leur nom et tel que spécifiée par les commentaires, les tests et autres documentations du code. L'**implémentation** d'une classe est le code contenu dans ses méthodes.

Java fait du concept d'interface une structure distincte, séparant expressément l'interface — ce qu'un objet doit faire — de l'implémentation — comment un objet remplit cet engagement. Les interfaces Java permettent à plusieurs classes d'offrir la même fonctionnalité et à une même classe d'implémenter plusieurs interfaces.

Plusieurs patterns de conception emploient les fonctionnalités intégrées à Java. Par exemple, vous pourriez utiliser une interface pour adapter l'interface d'une classe afin de répondre aux besoins d'un client en appliquant le pattern ADAPTER. Mais avant d'aborder certaines notions avancées, il peut être utile de s'assurer que vous maîtrisez les fonctionnalités de base, à commencer par les interfaces.

Interfaces et classes abstraites

Le livre original *Design Patterns* [Gamma et al. 1995] mentionne fréquemment l'emploi de classes abstraites mais pas du tout l'emploi d'interfaces. La raison en est que les langages C++ et Smalltalk, sur lesquels il s'appuie pour ses exemples, ne possèdent pas une telle structure. Cela ne remet toutefois pas en cause l'utilité de ce livre pour les développeurs Java, étant donné que les interfaces Java sont assez semblables aux classes abstraites.

Exercice 2.1

Enumérez trois différences entre les classes abstraites et les interfaces Java.

- *Les solutions des exercices de ce chapitre sont données dans l'Annexe B.*

Si les interfaces n'existaient pas, vous pourriez utiliser à la place des classes abstraites, comme dans C++. Les interfaces jouent toutefois un rôle essentiel dans le développement d'applications multiniveaux, ce qui justifie certainement leur statut particulier de structure distincte.

Considérez la définition d'une interface que les classes de simulation de fusée doivent implémenter. Les ingénieurs conçoivent toutes sortes de fusées, qu'elles soient à combustible solide ou liquide, avec des caractéristiques balistiques très diverses. Indépendamment de sa composition, la simulation d'une fusée doit fournir des chiffres pour la poussée (*thrust*) et la masse (*mass*). Voici le code qu'utilise Oozinoz pour définir l'interface de simulation de fusée :

```
package com.oozinoz.simulation;

public interface RocketSim {
    abstract double getMass();
    public double getThrust();
    void setSimTime(double t);
}
```

Exercice 2.2

Parmi les affirmations suivantes, lesquelles sont vraies ?

- A. Les méthodes de l'interface RocketSim sont toutes trois abstraites, même si seulement `getMass()` déclare cela explicitement.
- B. Les trois méthodes de l'interface sont publiques, même si seulement `getThrust()` déclare cela explicitement.
- C. L'interface est déclarée `public interface`, mais elle serait publique même si le mot clé `public` était omis.
- D. Il est possible de créer une autre interface, par exemple `RocketSimSolid`, qui étende `RocketSim`.
- E. Toute interface doit comporter au moins une méthode.
- F. Une interface peut déclarer des champs d'instance qu'une classe d'implémentation doit également déclarer.
- G. Bien qu'il ne soit pas possible d'instancier une interface, une interface peut déclarer des méthodes constructeurs dont la signature sera donnée par une classe d'implémentation.

Interfaces et obligations

Un avantage important des interfaces Java est qu'elles limitent l'interaction entre les objets. Cette limitation s'avère être un soulagement. En effet, une classe qui implémente une interface peut subir des changements considérables dans sa façon de remplir le contrat défini par l'interface sans que cela affecte aucunement ses clients.

Un développeur qui crée une classe implementant `RocketSim` a pour tâche d'écrire les méthodes `getMass()` et `getThrust()` qui retournent les mesures de performance d'une fusée. Autrement dit, il doit remplir le contrat de ces méthodes.

Parfois, les méthodes désignées par une interface n'ont aucune obligation de fournir un service à l'appelant. Dans certains cas, la classe d'implémentation peut même ignorer l'appel, implementant une méthode avec un corps vide.

Exercice 2.3

Donnez un exemple d'interface avec des méthodes n'impliquant aucune responsabilité pour la classe d'implémentation de retourner une valeur ou d'accomplir une quelconque action pour le compte de l'appelant.

Si vous créez une interface qui spécifie un ensemble de méthodes de notification, vous pourriez envisager d'utiliser une classe **stub**, c'est-à-dire une classe qui implémente l'interface avec des méthodes ne faisant rien. Les développeurs peuvent dériver des sous-classes de la classe stub, en redéfinissant uniquement les méthodes de l'interface qui sont importantes pour leur application. La classe `WindowAdapter` dans `java.awt.event` est un exemple d'une telle classe, comme illustré Figure 2.1 (pour une introduction rapide à UML, voyez l'Annexe D). Cette classe implémente toutes les méthodes de l'interface `WindowListener` mais les implémentations sont vides ; ces méthodes ne contiennent aucune instruction.

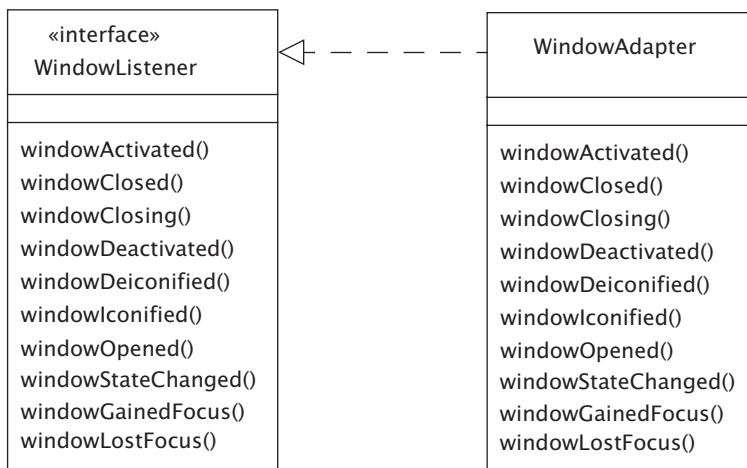


Figure 2.1

La classe `WindowAdapter` facilite l'enregistrement de listeners pour les événements de fenêtre en vous permettant d'ignorer ceux qui ne vous intéressent pas.

En plus de déclarer des méthodes, une interface peut déclarer des constantes. Dans l'exemple suivant, `ClassificationConstants` déclare deux constantes auxquelles les classes implémentant cette interface auront accès :

```
public interface ClassificationConstants {  
    static final int CONSUMER = 1;  
    static final int DISPLAY = 2;  
}
```

Une autre différence notable existe entre les interfaces et les classes abstraites. Tout en déclarant qu'elle étend (`extends`) une autre classe, une classe peut aussi déclarer qu'elle implémente (`implements`) une ou plusieurs interfaces.

Résumé

La puissance des interfaces réside dans le fait qu'elles stipulent ce qui est attendu et ce qui ne l'est pas en matière de collaboration entre classes. Elles sont semblables aux classes purement abstraites en ce qu'elles définissent des attentes mais ne les implémentent pas.

Maîtriser à la fois les concepts et les détails de l'application des interfaces Java demande du temps, mais le sacrifice en vaut la peine. Cette structure puissante est au cœur de nombreuses conceptions robustes et de plusieurs patterns de conception.

Au-delà des interfaces ordinaires

Vous pouvez simplifier et renforcer vos conceptions grâce à une application appropriée des interfaces Java. Parfois, cependant, la conception d'une interface doit dépasser sa définition et son utilisation ordinaires.

<i>Si vous envisagez de</i>	<i>Appliquez le pattern</i>
• Adapter l'interface d'une classe pour qu'elle corresponde à l'interface attendue par un client	ADAPTER
• Fournir une interface simple pour un ensemble de classes	FACADE
• Définir une interface qui s'applique à la fois à des objets individuels et à des groupes d'objets	COMPOSITE
• Découpler une abstraction de son implémentation de sorte que les deux puissent varier indépendamment	BRIDGE

L'objectif de chaque pattern de conception est de résoudre un problème dans un certain contexte. Les patterns d'interface conviennent dans des contextes où vous avez besoin de définir ou de redéfinir l'accès aux méthodes d'une classe ou d'un groupe de classes. Par exemple, lorsque vous disposez d'une classe qui accomplit un service nécessaire, mais dont les noms de méthodes ne correspondent pas aux attentes d'un client, vous pouvez appliquer le pattern ADAPTER.

3

ADAPTER

Un objet est un **client** lorsqu'il a besoin d'appeler votre code. Dans certains cas, votre code existe déjà et le développeur peut créer le client de manière à ce qu'il utilise les interfaces de vos objets. Dans d'autres, le client peut être développé indépendamment de votre code. Par exemple, un programme de simulation de fusée pourrait être conçu pour utiliser les informations techniques que vous fournissez, mais une telle simulation aurait sa propre définition du comportement que doit avoir une fusée. Si une classe existante est en mesure d'assurer les services requis par un client mais que ses noms de méthodes diffèrent, vous pouvez appliquer le pattern ADAPTER.

L'objectif du pattern ADAPTER est de fournir l'interface qu'un client attend en utilisant les services d'une classe dont l'interface est différente.

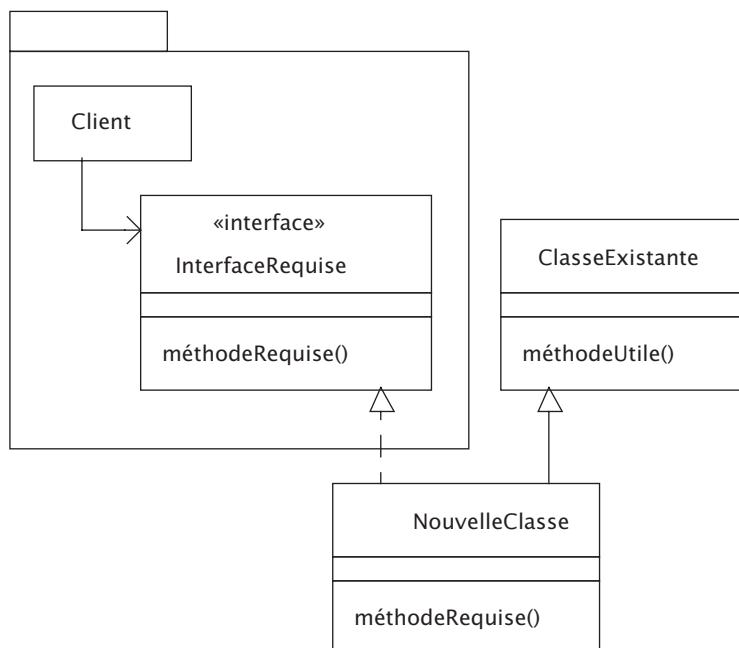
Adaptation à une interface

Le développeur d'un client peut avoir prévu les situations où vous aurez besoin d'adapter votre code au sien. Cela est évident s'il a fourni une interface qui définit les services dont le code client a besoin, comme dans l'exemple de la Figure 3.1. Une classe cliente invoque une méthode `méthodeRequise()` déclarée dans une interface. Supposez que vous avez trouvé une classe existante avec une méthode nommée par exemple `méthodeUtile()` capable de répondre aux besoins du client. Vous pouvez alors adapter cette classe au client en écrivant une classe qui étend `ClasseExistante`, implémente `InterfaceRequise` et redéfinit `méthodeRequise()` de sorte qu'elle délègue ses demandes à `méthodeUtile()`.

La classe `NouvelleClasse` est un exemple de ADAPTER. Une instance de cette classe est une instance de `InterfaceRequise`. En d'autres termes, `NouvelleClasse` répond aux besoins du client.

Figure 3.1

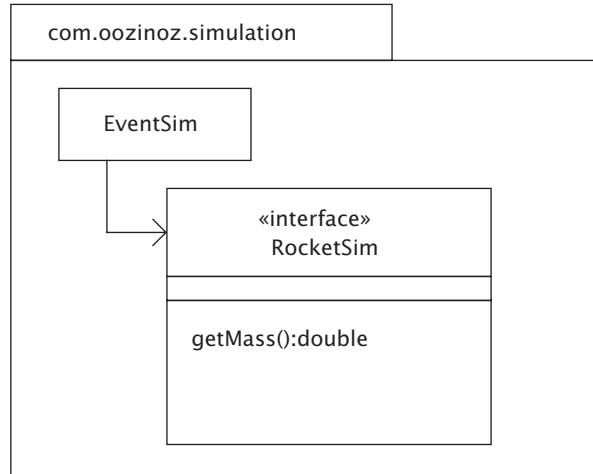
Lorsque le développeur du code client définit précisément les besoins du client, vous pouvez remplir le contrat défini par l'interface en adaptant le code existant.



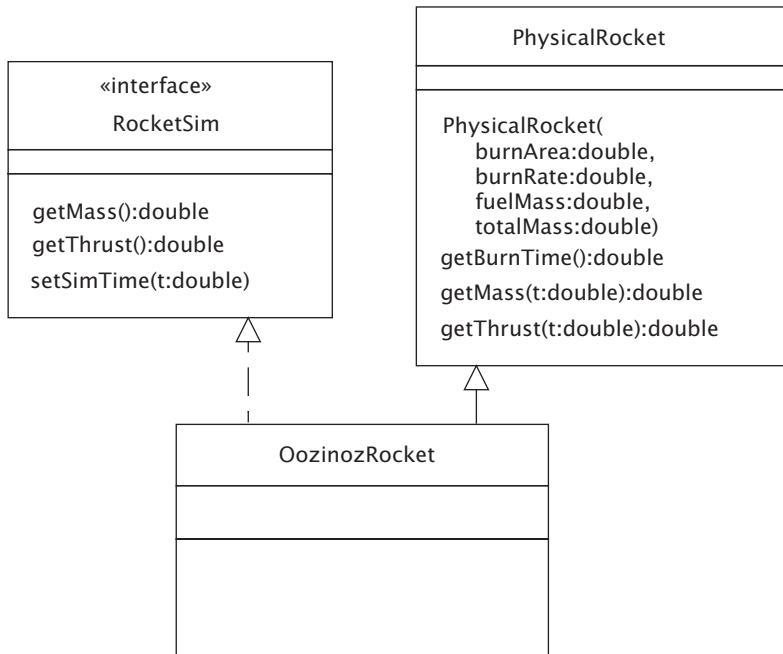
Pour prendre un exemple plus concret, imaginez que vous travaillez avec un package qui simule le vol et le minutage de fusées comme celles fabriquées par Oozinoz. Ce package inclut un simulateur d'événements qui couvre les effets du lancement de plusieurs fusées, ainsi qu'une interface qui spécifie le comportement d'une fusée. La Figure 3.2 illustre ce package.

Vous disposez d'une classe `PhysicalRocket` que vous voulez inclure dans la simulation. Cette classe possède des méthodes qui correspondent approximativement au comportement requis par le simulateur. Vous pouvez donc appliquer ADAPTER en dérivant de `PhysicalRocket` une sous-classe qui implémente l'interface `RocketSim`. La Figure 3.3 illustre partiellement cette conception.

La classe `PhysicalRocket` contient les informations dont le simulateur a besoin, mais ses méthodes ne correspondent pas exactement à celles que le programme de simulation déclare dans l'interface `RocketSim`. Cette différence tient au fait que le simulateur possède une horloge interne et actualise occasionnellement les objets simulés en invoquant une méthode `setSimTime()`. Pour adapter la classe `PhysicalRocket` aux exigences du simulateur, un objet `OozinozRocket` pourrait utiliser une variable d'instance `time` et la passer aux méthodes de la classe `PhysicalRocket` lorsque nécessaire.

**Figure 3.2**

Le package Simulation définit clairement ses exigences pour simuler le vol d'une fusée.

**Figure 3.3**

Une fois complété, ce diagramme représentera la conception d'une classe qui adapte la classe PhysicalRocket pour répondre aux exigences de l'interface RocketSim.

Exercice 3.1

Complétez le diagramme de la Figure 3.3 en faisant en sorte que la classe OozinozRocket permette à un objet PhysicalRocket de prendre part à une simulation en tant qu'objet RocketSim. Partez du principe que vous ne pouvez modifier ni RocketSim ni PhysicalRocket.

■ *Les solutions des exercices de ce chapitre sont données dans l'Annexe B.*

Le code de PhysicalRocket est un peu complexe car il réunit toutes les caractéristiques physiques dont se sert Oozinoz pour modéliser une fusée. Mais c'est exactement la logique que nous voulons réutiliser. La classe adaptateur OozinozRocket traduit simplement les appels pour utiliser les méthodes de sa super-classe. Le code de cette nouvelle sous-classe pourrait ressembler à ce qui suit :

```
package com.oozinoz.firework;
import com.oozinoz.simulation.*;

public class OozinozRocket
    extends PhysicalRocket implements RocketSim {
    private double time;

    public OozinozRocket(
        double burnArea, double burnRate,
        double fuelMass, double totalMass) {
        super(burnArea, burnRate, fuelMass, totalMass);
    }

    public double getMass() {
        // Exercice !
    }

    public double getThrust() {
        // Exercice !
    }

    public void setSimTime(double time) {
        this.time = time;
    }
}
```

Exercice 3.2

Complétez le code de la classe OozinozRocket en définissant les méthodes `getMass()` et `getThrust()`.

Lorsqu'un client définit ses attentes dans une interface, vous pouvez appliquer ADAPTER en fournissant une classe qui implémente cette interface et étend une classe existante. Il se peut aussi que vous puissiez appliquer ce pattern même en l'absence d'une telle interface, auquel cas il convient d'utiliser un adaptateur d'objet.

Adaptateurs de classe et d'objet

Les conceptions des Figures 3.1 et 3.3 sont des *adaptateurs de classe*, c'est-à-dire que l'adaptation procède de la dérivation de sous-classes. Dans une telle conception, la nouvelle classe adaptateur implémente l'interface désirée et étend une classe existante. Cette approche ne fonctionne pas toujours, notamment lorsque l'ensemble de méthodes que vous voulez adapter n'est pas spécifié dans une interface. Dans ce cas, vous pouvez créer un **adaptateur d'objet**, c'est-à-dire un adaptateur qui utilise la délégation plutôt que la dérivation de sous-classes. La Figure 3.4 illustre cette conception (comparez-la aux diagrammes précédents).

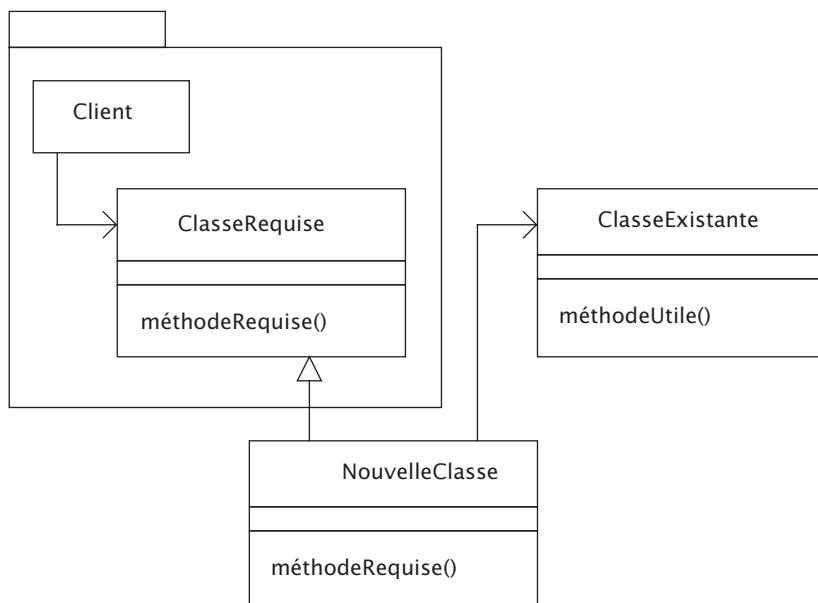


Figure 3.4

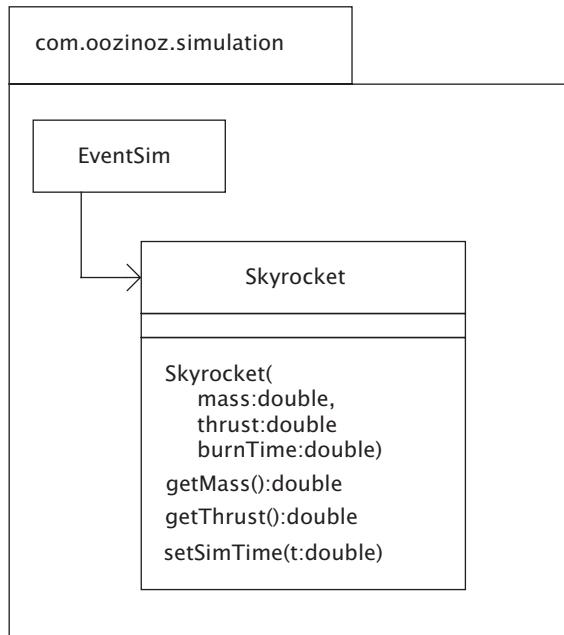
Vous pouvez créer un adaptateur d'objet en dérivant la sous-classe dont vous avez besoin et en remplissant les contrats des méthodes en vous appuyant sur un objet d'une classe existante.

La classe `NouvelleClasse` est un exemple de ADAPTER. Une instance de cette classe est une instance de `ClasseRequise`. En d'autres termes, `NouvelleClasse` répond aux besoins du client. Elle peut adapter la classe `ClasseExistante` pour satisfaire le client en utilisant une instance de cette classe.

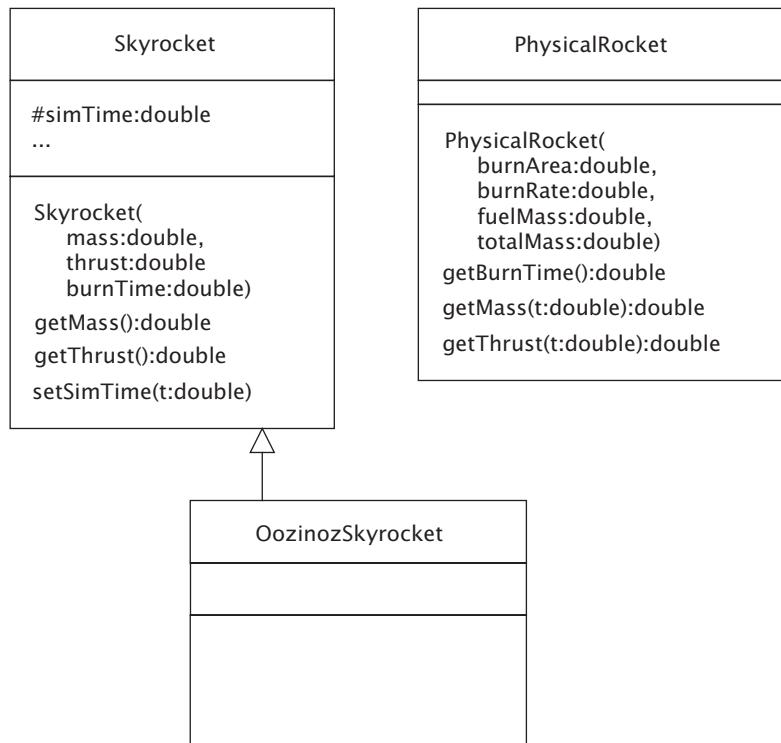
Pour prendre un exemple plus concret, imaginez que le package de simulation fonctionne directement avec une classe `Skyrocket`, sans spécifier d'interface définissant les comportements nécessaires pour la simulation. La Figure 3.5 illustre cette classe.

Figure 3.5

Dans cette conception-ci, le package com.oozinoz.simulation ne spécifie pas l'interface dont il a besoin pour modéliser une fusée.



La classe `Skyrocket` utilise un modèle physique assez rudimentaire. Par exemple, elle part du principe que la fusée se consume entièrement à mesure que son carburant brûle. Supposez que vous vouliez appliquer le modèle plus sophistiqué offert par la classe `PhysicalRocket` d'Oozinoz. Pour adapter la logique de cette classe à la simulation, vous pourriez créer une classe `OozinozSkyrocket` en tant qu'adaptateur d'objet qui étend `Skyrocket` et utilise un objet `PhysicalRocket`, comme le montre la Figure 3.6.

**Figure 3.6**

Une fois complété, ce diagramme représentera la conception d'un adaptateur d'objet qui s'appuie sur les informations d'une classe existante pour satisfaire le besoin d'un client d'utiliser un objet Skyrocket.

En tant qu'adaptateur d'objet, la classe **OozinozSkyrocket** étend **Skyrocket**, et non **PhysicalRocket**. Cela permet à un objet **OozinozSkyrocket** de servir de substitut chaque fois que le client requiert un objet **Skyrocket**. La classe **Skyrocket** supporte la dérivation de sous-classes en définissant sa variable **simTime** comme étant **protected**.

Exercice 3.3

Complétez le diagramme de la Figure 3.6 en faisant en sorte que des objets **OozinozSkyrocket** puissent servir d'objets **Skyrocket**.

Le code de la classe `OozinozSkyrocket` pourrait ressembler à ce qui suit :

```
package com.oozinoz.firework;
import com.oozinoz.simulation.*;

public class OozinozSkyrocket extends Skyrocket {
    private PhysicalRocket rocket;
    public OozinozSkyrocket(PhysicalRocket r) {
        super(
            r.getMass(),
            r.getThrust(),
            r.getBurnTime());
        rocket = r;
    }

    public double getMass() {
        return rocket.getMass(simTime);
    }

    public double getThrust() {
        return rocket.getThrust(simTime);
    }
}
```

La classe `OozinozSkyrocket` vous permet de fournir un objet `OozinozSkyrocket` chaque fois que le package requiert un objet `Skyrocket`. En général, les adaptateurs d'objet résolvent, partiellement du moins, le problème posé par l'adaptation d'un objet à une interface qui n'a pas été expressément définie.

Exercice 3.4

Citez une raison pour laquelle la conception d'adaptateur d'objet utilisée par la classe `OozinozSkyrocket` est plus fragile que l'approche avec adaptateur de classe.

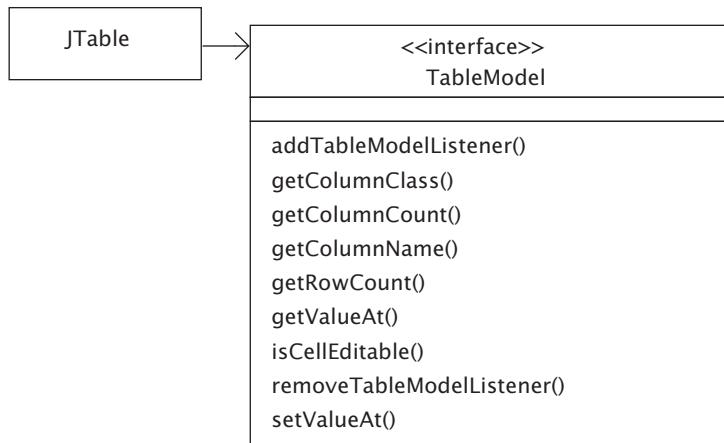
L'adaptateur d'objet pour la classe `Skyrocket` est une conception plus risquée que l'adaptateur de classe qui implémente l'interface `RocketSim`. Mais il ne faut pas trop se plaindre. Au moins, aucune méthode n'a été définie comme étant `final`, ce qui nous aurait empêchés de la redéfinir.

Adaptation de données pour un widget JTable

L'affichage de données sous forme de table donne lieu à un exemple courant d'adaptateur d'objet. Swing fournit le widget `JTable` pour afficher des tables. Les concepteurs de ce widget ne savaient naturellement pas quelles données il servirait à afficher. Aussi, plutôt que de coder en dur certaines structures de données, ils ont prévu une interface appelée `TableModel` (voir Figure 3.7) dont dépend le fonctionnement de `JTable`. Il vous revient ensuite de créer un adaptateur pour que vos données soient conformes à `TableModel`.

Figure 3.7

La classe `JTable` est un composant Swing qui affiche dans une table de GUI les données d'une implémentation de `TableModel`.



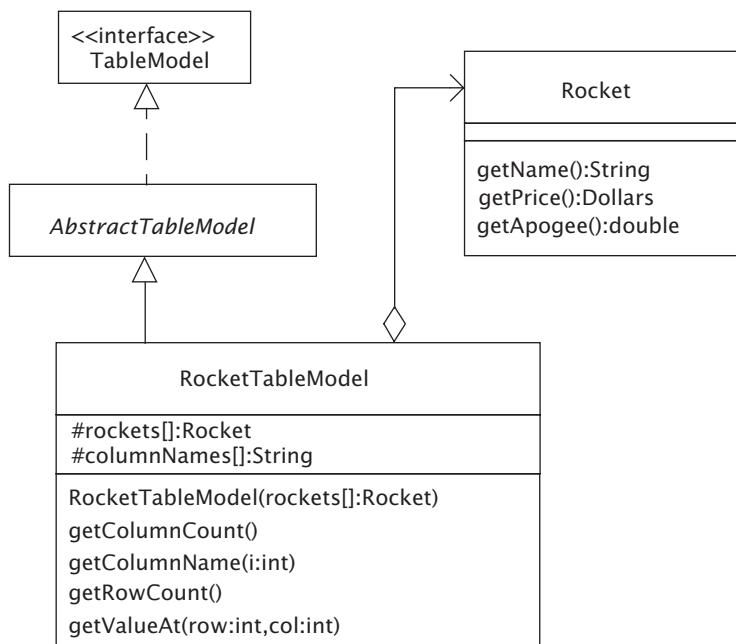
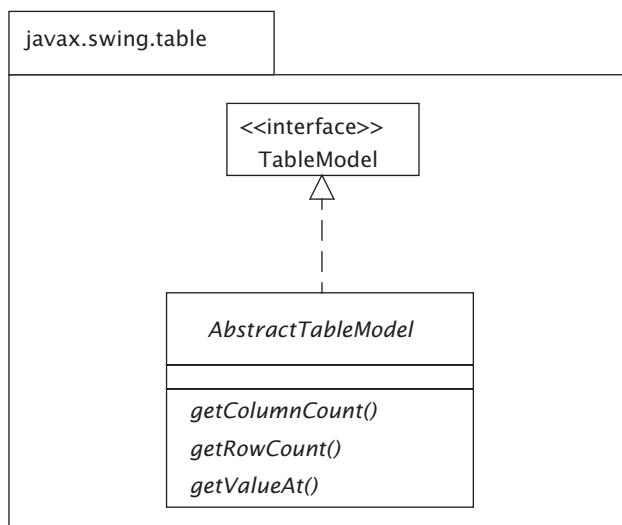
Nombre des méthodes de `TableModel` suggèrent la possibilité d'une implémentation par défaut. Heureusement, le **JDK** (*Java Development Kit*) inclut une classe abstraite qui fournit des implémentations par défaut pour toutes les méthodes de cette interface à l'exception de celles qui sont très spécifiques à un domaine. La Figure 3.8 illustre cette classe.

Imaginez que vous souhaitez lister quelques fusées dans une table en utilisant une interface utilisateur Swing. Comme le montre la Figure 3.9, vous pourriez créer une classe `RocketTableModel` qui adapte un tableau de fusées à l'interface attendue par `TableModel`.

La classe `RocketTableModel` doit étendre `AbstractTableModel` puisque cette dernière est une classe et non une interface. Lorsque l'interface cible de l'adaptation est supportée par une classe abstraite que vous souhaitez utiliser, vous devez

Figure 3.8

La classe `AbstractTableModel` prévoit des implémentations par défaut pour presque toutes les méthodes de `TableModel`.

**Figure 3.9**

La classe `RocketTableModel` adapte l'interface `TableModel` à la classe `Rocket` du domaine Oozinoz.

créer un adaptateur d'objet. Dans notre exemple, une autre raison qui justifie de ne pas recourir à un adaptateur de classe est que `RocketTableModel` n'est ni un type ni un sous-type de `Rocket`. Lorsqu'une classe adaptateur doit tirer ses informations de plusieurs objets, elle est habituellement implémentée en tant qu'adaptateur d'objet.

Retenez la différence : un adaptateur de classe étend une classe existante et implémente une interface cible tandis qu'un adaptateur d'objet étend une classe cible et délégué à une classe existante.

Une fois la classe `RocketTableModel` créée, vous pouvez facilement afficher des informations sur les fusées dans un objet Swing `JTable`, comme illustré Figure 3.10.

Figure 3.10

Une instance de `JTable` contenant des données sur les fusées.

Name	Price	Apogee
Shooter	\$3.95	50.0
Orbit	\$29.03	5000.0

```
package app.adapter;
import javax.swing.table.*;
import com.oozinoz.firework.Rocket;

public class RocketTableModel extends AbstractTableModel {
    protected Rocket[] rockets;
    protected String[] columnNames =
        new String[] { "Name", "Price", "Apogee" };

    public RocketTableModel(Rocket[] rockets) {
        this.rockets = rockets;
    }

    public int getColumnCount() {
        // Exercice !
    }

    public String getColumnName(int i) {
        // Exercice !
    }

    public int getRowCount() {
        // Exercice !
    }

    public Object getValueAt(int row, int col) {
        // Exercice !
    }
}
```

Exercice 3.5

Complétez le code des méthodes de `RocketTableModel` qui adaptent un tableau d'objets `Rocket` pour qu'il serve d'interface `TableModel`.

Pour obtenir le résultat de la Figure 3.10, vous pouvez créer deux objets fusée, les placer dans un tableau, créer une instance de `RocketTableModel` à partir du tableau, et utiliser des classes Swing pour afficher ce dernier. La classe `ShowRocketTable` en donne un exemple :

```
package app.adapter;

import java.awt.Component;
import java.awt.Font;

import javax.swing.*;

import com.oozinoz.firework.Rocket;
import com.oozinoz.utility.Dollars;

public class ShowRocketTable {
    public static void main(String[] args) {
        setFonts();
        JTable table = new JTable(getRocketTable());
        table.setRowHeight(36);
        JScrollPane pane = new JScrollPane(table);
        pane.setPreferredSize(
            new java.awt.Dimension(300, 100));
        display(pane, "Rockets");
    }

    public static void display(Component c, String title) {
        JFrame frame = new JFrame(title);
        frame.getContentPane().add(c);
        frame.setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }

    private static RocketTableModel getRocketTable() {
        Rocket r1 = new Rocket(
            "Shooter", 1.0, new Dollars(3.95), 50.0, 4.5);
        Rocket r2 = new Rocket(
            "Orbit", 2.0, new Dollars(29.03), 5000, 3.2);
        return new RocketTableModel(new Rocket[] { r1, r2 });
    }
}
```

```

private static void setFonts() {
    Font font = new Font("Dialog", Font.PLAIN, 18);
    UIManager.put("Table.font", font);
    UIManager.put("TableHeader.font", font);
}
}

```

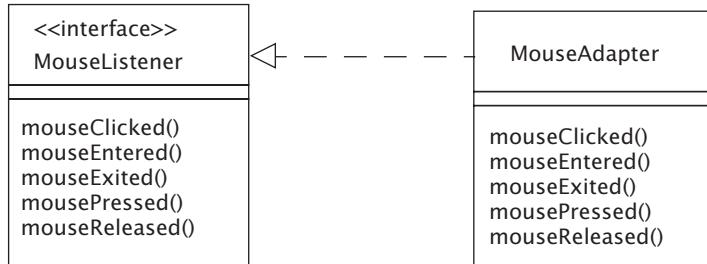
La classe `ShowRocketTable`, constituée elle-même de moins de vingt instructions, figure au-dessus de milliers d'autres instructions qui collaborent pour produire un composant table au sein d'un environnement GUI (*Graphical User Interface*). La classe `JTable` peut gérer pratiquement tous les aspects de l'affichage d'une table mais ne peut savoir à l'avance quelles données vous voudrez présenter. Pour vous permettre de fournir les données dont elle a besoin, elle vous donne la possibilité d'appliquer le pattern ADAPTER. Pour utiliser `JTable`, vous implémentez l'interface `TableModel` qu'elle attend, ainsi qu'une classe fournissant les données à afficher.

Identification d'adaptateurs

Le Chapitre 2 a évoqué l'intérêt que présente la classe `WindowAdapter`. La classe `MouseAdapter` illustrée Figure 3.11 est un autre exemple de classe stub (c'est-à-dire qui ne définit pas les méthodes requises par l'interface qu'elle implémente).

Figure 3.11

La classe `MouseAdapter` implémente la classe `MouseListener` en laissant vide le corps de ses méthodes.



Exercice 3.6

Pouvez-vous considérer que vous appliquez le pattern ADAPTER lorsque vous utilisez la classe `MouseAdapter` ? Expliquez votre réponse.

Résumé

Le pattern ADAPTER vous permet d'utiliser une classe existante pour répondre aux exigences d'une classe cliente. Lorsqu'un client spécifie ses exigences dans une interface, vous pouvez généralement créer une nouvelle classe qui implémente l'interface et étend la classe existante. Cette approche produit un adaptateur de classe qui traduit les appels du client en appels des méthodes de la classe existante.

Lorsque le client ne spécifie pas l'interface dont il a besoin, vous pouvez quand même appliquer ADAPTER en créant une sous-classe cliente qui utilise une instance de la classe existante. Cette approche produit un adaptateur d'objet qui transmet les appels du client à cette instance. Elle n'est pas dénuée de risques, surtout si vous omettez (ou êtes dans l'impossibilité) de redéfinir toutes les méthodes que le client pourrait appeler.

Le composant `JTable` dans Swing est un bon exemple de classe à laquelle ses concepteurs ont appliqué le pattern ADAPTER. Il se présente en tant que client ayant besoin des informations de table telles que définies par l'interface `TableModel`. Il vous est ainsi plus facile d'écrire un adaptateur qui alimente la table en données à partir d'objets du domaine, tels que des instances de la classe `Rocket`.

Pour utiliser `JTable`, on crée souvent un adaptateur d'objet qui délègue les appels aux instances d'une classe existante. Deux aspects de `JTable` font qu'il est peu probable qu'un adaptateur de classe soit utilisé. Premièrement, l'adaptateur est habituellement créé en étendant `AbstractTableModel`, auquel cas il n'est pas possible d'étendre également la classe existante. Deuxièmement, la classe `JTable` requiert un ensemble d'objets, et un adaptateur d'objet convient mieux pour adapter des informations tirées de plusieurs objets.

Lorsque vous concevez vos systèmes, considérez la puissance et la souplesse offertes par une architecture qui tire parti de ADAPTER.

4

FACADE

Un gros avantage de la POO est qu'elle permet d'éviter le développement de programmes monolithiques au code irrémédiablement enchevêtré. Dans un système OO, une **application** est, idéalement, une classe minimale qui unit les comportements d'autres classes groupées en kits d'outils réutilisables. Un développeur de kits d'outils ou de sous-systèmes crée souvent des packages de classes bien conçues sans fournir d'applications les liant. Les packages dans les bibliothèques de classes Java se présentent généralement ainsi. Ce sont des kits d'outils à partir desquels vous pouvez tisser une variété infinie d'applications spécifiques.

La réutilisabilité des kits d'outils s'accompagne d'un inconvénient : l'applicabilité diverse des classes dans un sous-système OO met à la disposition du développeur une quantité tellement impressionnante d'options qu'il lui est parfois difficile de savoir par où commencer. Un environnement de développement intégré, ou IDE (*Integrated Development Environment*), tel qu'Eclipse, peut affranchir le développeur d'une certaine part de la complexité du kit, mais il ajoute en revanche une grande quantité de code que le développeur ne souhaitera pas forcément maintenir.

Une autre approche pour simplifier l'emploi d'un kit d'outils est de fournir une façade — une petite quantité de code qui permet un usage typique à peu de frais des classes de la bibliothèque. Une façade est elle-même une classe avec un niveau de fonctionnalités situé entre le kit d'outils et une application complète, proposant un emploi simplifié des classes d'un package ou d'un sous-système.

L'objectif du pattern FACADE est de fournir une interface simplifiant l'emploi d'un sous-système.

Façades, utilitaires et démos

Une classe de façade peut ne contenir que des méthodes statiques, auquel cas elle est appelée un **utilitaire** dans *UML (Guide de l'utilisateur UML)* [Booch, Rumbaugh, et Jacobson 1999]. Nous introduirons par la suite une classe UI (*User Interface*), qui aurait pu recevoir seulement des méthodes statiques, bien que procéder ainsi aurait empêché par la suite la redéfinition des méthodes dans les sous-classes.

Une **démo** est un exemple qui montre comment employer une classe ou un sous-système. A cet égard, la valeur des démos peut être vue comme étant égale à celle des façades.

Exercice 4.1

Indiquez deux différences entre une démo et une façade.

- *Les solutions des exercices de ce chapitre sont données dans l'Annexe B.*

Le package `javax.swing` contient `JOptionPane`, une classe qui permet d'afficher facilement une boîte de dialogue standard. Par exemple, le code suivant affiche et réaffiche une boîte de dialogue jusqu'à ce que l'utilisateur clique sur le bouton Yes, comme illustré Figure 4.1.

```
package app.facade;

import javax.swing.*;
import java.awt.Font;

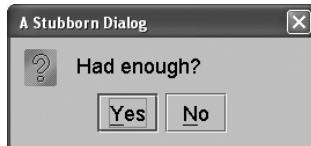
public class ShowOptionPane {
    public static void main(String[] args) {
        Font font = new Font("Dialog", Font.PLAIN, 18);
        UIManager.put("Button.font", font);
        UIManager.put("Label.font", font);

        int option;
        do {
            option = JOptionPane.showConfirmDialog(
                null,
                "Had enough?",
                "Question",
                JOptionPane.YES_NO_OPTION);
        } while (option == JOptionPane.NO_OPTION);
    }
}
```

```
        "A Stubborn Dialog",
        JOptionPane.YES_NO_OPTION);
    } while (option == JOptionPane.NO_OPTION);
}
}
```

Figure 4.1

La classe JOptionPane facilite l'affichage de boîtes de dialogue.



Exercice 4.2

La classe JOptionPane facilite l'affichage d'une boîte de dialogue. Indiquez si cette classe est une façade, un utilitaire ou une démo. Justifiez votre réponse.

Exercice 4.3

Peu de façades apparaissent dans les bibliothèques de classes Java. Pour quelle raison ?

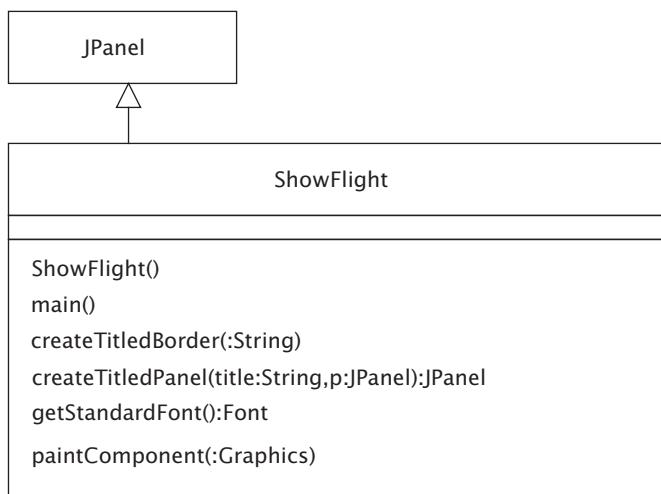
Refactorisation pour appliquer FAÇADE

Les façades sont souvent introduites hors de la phase de développement normal d'une application. Lors de la tâche de séparation des problèmes dans votre code en diverses classes, vous pouvez **refactoriser**, ou restructurer, le système en extrayant une classe dont la tâche principale est de fournir un accès simplifié à un sous-système. Considérez un exemple remontant aux premiers jours d'Oozinoz, où aucun standard de développement de GUI n'avait encore été adopté. Supposez que vous vous retrouviez à examiner une application qu'un développeur a créée pour afficher la trajectoire d'une bombe aérienne n'ayant pas explosé. La Figure 4.2 illustre cette classe.

Les bombes sont prévues pour exploser très haut dans le ciel en produisant des effets spectaculaires. Parfois, une bombe n'explose pas du tout. Dans ce cas, son retour sur terre devient intéressant. A la différence d'une fusée, une bombe n'est pas auto-propulsée. Aussi, si vous ignorez les effets dus au vent et à la résistance de l'air, la trajectoire d'une bombe ayant un raté est une simple parabole.

Figure 4.2

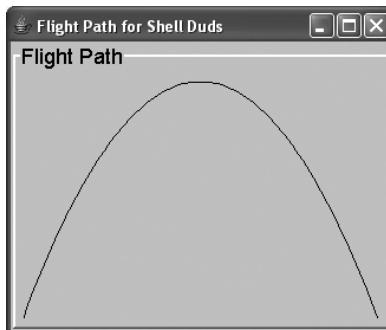
La classe ShowFlight affiche la trajectoire d'une bombe aérienne ayant un raté.



La Figure 4.3 illustre une capture d'écran de la fenêtre qui apparaît lorsque vous exécutez `ShowFlight.main()`.

Figure 4.3

L'application ShowFlight montre l'endroit où une bombe qui n'a pas explosé retombe.



La classe `ShowFlight` présente un problème : elle mêle trois objectifs. Son objectif principal est d'agir en tant que panneau d'affichage d'une trajectoire. Un deuxième objectif de cette classe est d'agir en tant qu'application complète, incorporant et affichant le panneau de trajectoire dans un cadre composé d'un titre. Enfin, son dernier objectif est de calculer la trajectoire parabolique que suit la bombe défaillante, le calcul étant réalisé dans `paintComponent()` :

```

protected void paintComponent(Graphics g) {
    super.paintComponent(g); // dessine l'arrière-plan
  
```

```
int nPoint = 101;
double w = getWidth() - 1;
double h = getHeight() - 1;
int[] x = new int[nPoint];
int[] y = new int[nPoint];
for (int i = 0; i < nPoint; i++) {
    // t va de 0 à 1
    double t = ((double) i) / (nPoint - 1);
    // x va de 0 à w
    x[i] = (int) (t * w);
    // y est h pour t = 0 et t = 1, et 0 pour t = 0,5
    y[i] = (int) (4 * h * (t - .5) * (t - .5));
}
g.drawPolyline(x, y, nPoint);
```

Voyez l'encadré intitulé "Equations paramétriques" plus loin dans ce chapitre pour une explication de la façon dont le code définit les valeurs x et y de la trajectoire.

Il n'est pas nécessaire d'avoir un constructeur. Il existe des méthodes statiques utilitaires qui permettent d'incorporer un titre dans un cadre et de définir une police standard.

```
public static TitledBorder createTitledBorder(String title){
    TitledBorder tb = BorderFactory.createTitledBorder(
        BorderFactory.createBevelBorder(BevelBorder.RAISED),
        title,
        TitledBorder.LEFT,
        TitledBorder.TOP);
    tb.setTitleColor(Color.black);
    tb.setFont(getStandardFont());
    return tb;
}
public static JPanel createTitledBorderPanel(
    String title, JPanel in) {
    JPanel out = new JPanel();
    out.add(in);
    out.setBorder(createTitledBorder(title));
    return out;
}
public static Font getStandardFont() {
    return new Font("Dialog", Font.PLAIN, 18);
}
```

Notez que la méthode `createTitledBorderPanel()` place le composant reçu à l'intérieur d'une bordure en relief pour produire un léger espace de remplissage, empêchant la courbe de la trajectoire de toucher les bords du panneau. La méthode `main()` ajoute

aussi à l'objet de formulaire un espace de remplissage qu'il utilise pour contenir les composants de l'application :

```
public static void main(String[] args) {
    ShowFlight flight = new ShowFlight();
    flight.setPreferredSize(new Dimension(300, 200));
    JPanel panel = createTitledPanel("Flight Path", flight);

    JFrame frame = new JFrame("Flight Path for Shell Duds");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(panel);

    frame.pack();
    frame.setVisible(true);
}
```

L'exécution de ce programme produit la fenêtre illustrée Figure 4.3.

Équations paramétriques

Lorsque vous devez dessiner une courbe, il peut être difficile de décrire des valeurs y en tant que fonctions de valeurs x. Les **équations paramétriques** permettent de définir ces deux types de valeurs en fonction d'un troisième paramètre. Plus spécifiquement, vous pouvez définir un temps t allant de 0 à 1 alors que la courbe est dessinée, et définir x et y en tant que fonctions du paramètre t.

Par exemple, supposez que le tracé de la trajectoire parabolique doive s'étendre sur la largeur w d'un objet Graphics. Une équation paramétrique pour x est simple :

$$x = w * t$$

Notez que pendant que t passe de 0 à 1, x va de 0 à w.

Les valeurs y d'une parabole doivent varier avec le carré de la valeur de t, et les valeurs de y doivent augmenter en allant vers le bas de l'écran. Pour une trajectoire parabolique, la valeur y devrait être égale à 0 au temps t = 0,5. Aussi pouvons-nous écrire l'équation initiale comme suit :

$$y = k * (t - 0,5) * (t - 0,5)$$

Ici, k représente une constante que nous devons encore déterminer. L'équation prévoit y à 0 lorsque t = 0,5, et avec une valeur identique pour t = 0 et t = 1. A ces deux instants t, y devrait être égale à h, la hauteur de la zone d'affichage. Avec un peu de manipulation algébrique, vous pouvez trouver l'équation complète pour y :

$$y = 4 * h * (t - 0,5) * (t - 0,5)$$

La Figure 4.3 illustre le résultat des équations en action.

Un autre avantage des équations paramétriques est qu'elles ne posent pas de problème pour dessiner des courbes qui possèdent plus d'une valeur y pour une valeur x. Considérez le dessin d'un cercle. L'équation d'un cercle avec un rayon de 1 est posée comme suit :

$$x^2 + y^2 = r^2$$

ou :

$$y = \pm \sqrt{r^2 - x^2}$$

Devoir gérer le fait que deux valeurs y sont produites pour chaque valeur x est compliqué. Il est aussi difficile d'ajuster ces valeurs pour dessiner correctement la courbe à l'intérieur des dimensions h (hauteur) et w (largeur) d'un objet Graphics. Les coordonnées polaires simplifient la fonction pour un cercle :

$$\begin{aligned}x &= r * \cos(\theta) \\y &= r * \sin(\theta)\end{aligned}$$

Ces formules sont des équations paramétriques qui définissent x et y en tant que fonctions d'un nouveau paramètre θ . La variable θ représente la courbure d'un arc qui varie de 0 à $2 * \pi$ alors que le cercle est dessiné. Vous pouvez définir le rayon d'un cercle de manière qu'il s'inscrive à l'intérieur des dimensions d'un objet Graphics. Quelques équations paramétriques suffisent pour dessiner un cercle dans les limites d'un tel objet, comme le montre l'exemple suivant :

```
theta = 2 * pi * t
r = min(w, h)/2
x = w/2 + r * cos(theta)
y = h/2 - r * sin(theta)
```

La transposition de ces équations dans le code produit le cercle illustré Figure 4.4 — le code qui produit cet affichage se trouve dans l'application ShowCircle sur le site oozinoz.com.

Le code dessinant un cercle est une transposition relativement directe des formules mathématiques. Il y a toutefois une subtilité dans ce sens que le code réduit la hauteur et la largeur de l'objet Graphics car les pixels sont numérotés de 0 à $h - 1$ et de 0 à $w - 1$.

```
package app.facade;

import javax.swing.*;
import java.awt.*;

import com.oozinoz.ui.SwingFacade;

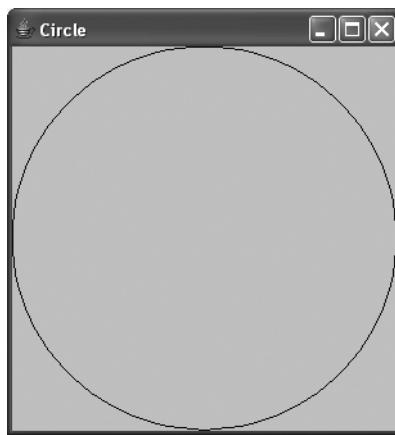
public class ShowCircle extends JPanel {
    public static void main(String[] args) {
        ShowCircle sc = new ShowCircle();
        sc.setPreferredSize(new Dimension(300, 300));
        SwingFacade.launch(sc, "Circle");
    }
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        int nPoint = 101;
        double w = getWidth() - 1;
        double h = getHeight() - 1;
        double r = Math.min(w, h) / 2.0;
        int[] x = new int[nPoint];
        int[] y = new int[nPoint];
        for (int i = 0; i < nPoint; i++) {
            double t = ((double) i) / (nPoint - 1);
            double theta = Math.PI * 2.0 * t;
            x[i] = (int) (w / 2 + r * Math.cos(theta));
            y[i] = (int) (h / 2 - r * Math.sin(theta));
        }
    }
}
```

```
        }
        g.drawPolyline(x, y, nPoint);
    }
}
```

Exprimer les fonctions *x* et *y* par rapport à *t* vous permet de diviser les tâches de détermination des valeurs *x* et *y*. C'est souvent plus simple que de devoir définir *y* en fonction de *x* et cela facilite souvent la transposition de *x* et de *y* en coordonnées d'un objet *Graphics*. Les équations paramétriques simplifient également le dessin de courbes où *y* n'est pas une fonction monovaluée de *x*.

Figure 4.4

Les équations paramétriques simplifient la modélisation de courbes lorsque y n'est pas une fonction monovaluée de x.



Le code de la classe *ShowFlight* fonctionne, mais vous pouvez le rendre plus facile à maintenir et plus réutilisable en le retravaillant pour créer des classes se concentrant sur des problèmes distincts. Supposez qu'après une révision du code, vous décidiez :

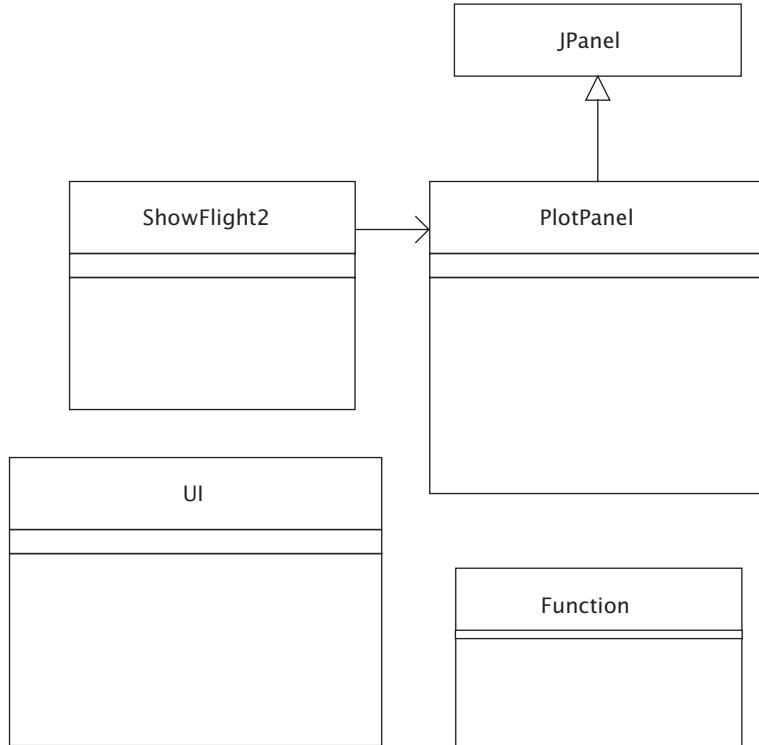
- D'introduire une classe *Function* avec une méthode *f()* qui accepte un type *double* (une valeur de temps) et retourne un *double* (la valeur de la fonction).
- De déplacer le code dessinant la courbe de la classe *ShowFlight* vers une classe *PlotPanel1*, mais de le modifier pour qu'il utilise des objets *Function* pour les valeurs *x* et *y*. Définissez le constructeur *PlotPanel1* de manière qu'il accepte deux instances de *Function* ainsi que le nombre de points à dessiner.
- De déplacer la méthode *createTitledBorder()* vers la classe utilitaire *UI* pour construire un panneau avec un titre, comme le fait déjà la classe *ShowFlight*.

Exercice 4.4

Complétez le diagramme de la Figure 4.5 pour présenter le code de ShowFlight réparti en trois types : une classe Function, une classe PlotPanel qui dessine deux fonctions paramétriques, et une classe de façade UI. Dans votre nouvelle conception, faites en sorte que ShowFlight2 crée un objet Function pour les valeurs y et incorpore une méthode main() qui lance l'application.

Figure 4.5

L'application de dessin d'une trajectoire parabolique restructurée en trois classes s'acquittant chacune d'une tâche.



Après ces changements, la classe Function définit l'apparence des équations paramétriques. Supposez que vous créez un package com.oozinoz.function pour contenir la classe Function et d'autres types. Le cœur de Function.java pourrait être :

```
public abstract double f(double t);
```

La classe `PlotPanel` résultant de la restructuration du code n'a qu'un travail à réaliser : afficher une paire d'équations paramétriques :

```
package com.oozinoz.ui;

import java.awt.Color;
import java.awt.Graphics;

import javax.swing.JPanel;
import com.oozinoz.function.Function;

public class PlotPanel extends JPanel {
    private int points;
    private int[] xPoints;
    private int[] yPoints;

    private Function xFunction;
    private Function yFunction;

    public PlotPanel(
        int nPoint, Function xFunc, Function yFunc) {
        points = nPoint;
        xPoints = new int[points];
        yPoints = new int[points];
        xFunction = xFunc;
        yFunction = yFunc;
        setBackground(Color.WHITE);
    }

    protected void paintComponent(Graphics graphics) {
        double w = getWidth() - 1;
        double h = getHeight() - 1;

        for (int i = 0; i < points; i++) {
            double t = ((double) i) / (points - 1);
            xPoints[i] = (int) (xFunction.f(t) * w);
            yPoints[i] = (int) (h * (1 - yFunction.f(t)));
        }

        graphics.drawPolyline(xPoints, yPoints, points);
    }
}
```

Notez que la classe `PlotPanel` fait maintenant partie du package `com.oozinoz.ui`, où réside aussi la classe `UI`. Après restructuration de la classe `ShowFlight`, la classe `UI` inclut aussi les méthodes `createTitledPanel()` et `createTitledBorder()`. La classe `UI` se transforme en façade qui facilite l'emploi de composants graphiques Java.

Une application qui utiliserait ces composants pourrait être une petite classe ayant pour seule tâche de les mettre en place et de les afficher. Par exemple, le code de la classe ShowFlight2 se présente comme suit :

```
package app.facade;

import java.awt.Dimension;
import javax.swing.JFrame;
import com.oozinoz.function.Function;
import com.oozinoz.function.T;
import com.oozinoz.ui.PlotPanel;
import com.oozinoz.ui.UI;

public class ShowFlight2 {
    public static void main(String[] args) {
        PlotPanel p = new PlotPanel(
            101,
            new T(),
            new ShowFlight2().new YFunction());
        p.setPreferredSize(new Dimension(300, 200));

        JFrame frame = new JFrame(
            "Flight Path for Shell Duds");
        frame.setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(
            UI.NORMAL.createTitledPanel("Flight Path", p));

        frame.pack();
        frame.setVisible(true);
    }

    private class YFunction extends Function {
        public YFunction() {
            super(new Function[] {});
        }

        public double f(double t) {
            // y est 0 pour t = 0 et 1 ; y est 1 pour t = 0,5
            return 4 * t * (1 - t);
        }
    }
}
```

La classe ShowFlight2 fournit la classe YFunction pour la trajectoire. La méthode `main()` met en place l'interface utilisateur et l'affiche. L'exécution de cette classe produit les mêmes résultats que la classe ShowFlight originale. La différence est que vous disposez maintenant d'une façade réutilisable qui simplifie la création d'une interface utilisateur graphique dans des applications Java.

Résumé

D'ordinaire, vous devriez refactoriser les classes d'un sous-système jusqu'à ce que chaque classe ait un objectif spécifique bien défini. Cette approche permet d'obtenir un code plus facile à maintenir. Il est toutefois possible qu'un utilisateur de votre sous-système puisse éprouver des difficultés pour trouver par où commencer. Pour pallier cet inconvénient et aider le développeur exploitant votre code, vous pouvez fournir des démos ou des façades avec votre sous-système. Une démo est généralement autonome, c'est une application non réutilisable qui montre une façon d'appliquer un sous-système. Une façade est une classe configurable et réutilisable, avec une interface de plus haut niveau qui simplifie l'emploi du sous-système.

5

COMPOSITE

Un **COMPOSITE** est un groupe d’objets contenant aussi bien des éléments individuels que des éléments contenant d’autres objets. Certains objets contenus représentent donc eux-mêmes des groupes et d’autres sont des objets individuels appelés des **feuilles** (*leaf*). Lorsque vous modélez un objet composite, deux concepts efficaces émergent. Une première idée importante est de concevoir des groupes de manière à englober des éléments individuels ou d’autres groupes — une erreur fréquente est de définir des groupes ne contenant que des feuilles. Un autre concept puissant est la définition de comportements communs aux deux types d’objets, individuels et composites. Vous pouvez unir ces deux idées en définissant un type commun aux groupes et aux feuilles, et en modélisant des groupes de façon qu’ils contiennent un ensemble d’objets de ce type.

L’objectif du pattern COMPOSITE est de permettre aux clients de traiter de façon uniforme des objets individuels et des compositions d’objets.

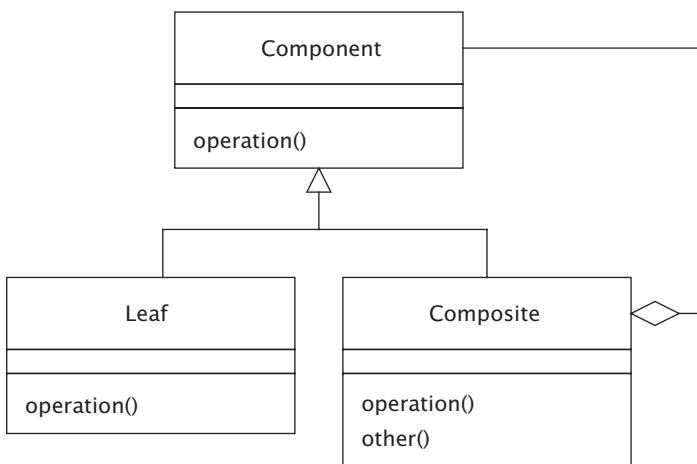
Un composite ordinaire

La Figure 5.1 illustre une structure composite ordinaire. Les classes **Leaf** et **Composite** partagent une interface commune, **Component**. Un objet **Composite** sous-tend d’autres objets **Composite** et **Leaf**.

Notez que, dans la Figure 5.1, **Component** est une classe abstraite sans opérations concrètes. Vous pouvez donc la définir en tant qu’interface implémentée par **Leaf** et **Composite**.

Figure 5.1

Les concepts essentiels véhiculés par le pattern COMPOSITE sont qu'un objet composite peut aussi contenir, outre des feuilles, d'autres objets composites, et que les nœuds composites et feuilles partagent une interface commune.



Exercice 5.1

Pourquoi la classe Composite dans la Figure 5.1 sous-tend-elle un ensemble d'objets Component et pas simplement un ensemble de feuilles ?

- *Les solutions des exercices de ce chapitre sont données dans l'Annexe B.*

Comportement récursif dans les objets composites

Les ingénieurs d'Oozinoz ont perçu une composition naturelle dans les machines qu'ils utilisent pour la production de pièces d'artifice. Une unité de production se compose de travées, chaque travée contient une ou plusieurs lignes de montage, et chaque ligne comprend un ensemble de machines qui collaborent pour produire des pièces et respecter un calendrier. Les développeurs ont modélisé ce domaine en traitant unités de production, travées et lignes de montage comme des "machines" composites, en utilisant le diagramme de classes présenté à la Figure 5.2.

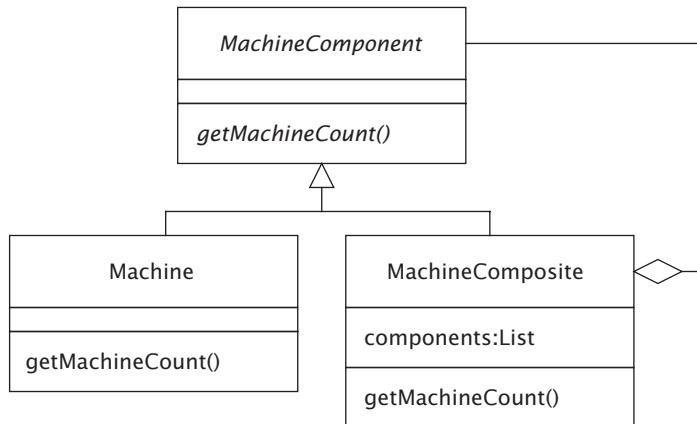
Comme le montre la figure, un comportement qui s'applique à la fois aux machines individuelles et aux groupes de machines est `getMachineCount()`, qui retourne le nombre de machines pour un composant donné.

Exercice 5.2

Ecrivez le code des méthodes `getMachineCount()` implémentées respectivement par `Machine` et `MachineComposite`.

Figure 5.2

La méthode `getMachineCount()` est un comportement approprié aussi bien pour les machines individuelles que pour les machines composites.



Supposez que nous envisagions l'ajout des méthodes suivantes dans `MachineComponent` :

Méthode	Comportement
<code>isCompletelyUp()</code>	Indique si toutes les machines d'un composant se trouvent dans un état actif
<code>stopAll()</code>	Ordonne à toutes les machines d'un composant d'arrêter leur travail
<code>getOwners()</code>	Retourne un ensemble d'ingénieurs des méthodes responsables des machines d'un composant
<code>getMaterial()</code>	Retourne tous les produits en cours de traitement dans un composant-machine

Le fonctionnement de chaque méthode dans `MachineComponent` est récursif. Par exemple, le compte de machines dans un objet composite est le total des comptes de machines de ses composants.

Exercice 5.3

Pour chaque méthode déclarée par `MachineComponent`, donnez une définition récursive pour `MachineComposite` et non récursive pour `Machine`.

<i>Méthode</i>	<i>Classe</i>	<i>Définition</i>
<code>getMachineCount()</code>	<code>MachineComposite</code>	Retourne la somme des comptes pour chaque composant de <code>Component</code>
	<code>Machine</code>	Retourne 1
<code>isCompletelyUp()</code>	<code>MachineComposite</code>	??
	<code>Machine</code>	??
<code>stopAll()</code>	<code>MachineComposite</code>	??
	<code>Machine</code>	??
<code>getOwners()</code>	<code>MachineComposite</code>	??
	<code>Machine</code>	??
<code>getMaterial()</code>	<code>MachineComposite</code>	??
	<code>Machine</code>	??

Objets composites, arbres et cycles

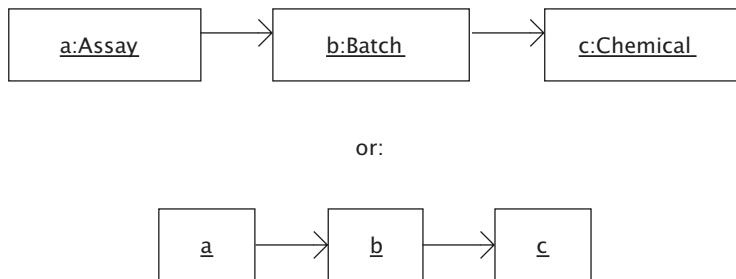
Dans une structure composite, nous pouvons dire qu'un nœud est un arbre s'il contient des références à d'autres nœuds. Cette définition est cependant trop vague. Pour être plus précis, nous pouvons appliquer quelques termes de la théorie des graphes à la modélisation d'objets. Nous pouvons commencer par dessiner un modèle objet sous forme d'un graphe — un ensemble de nœuds et d'arêtes — avec des objets en tant que nœuds et des références d'objet en tant qu'arêtes.

Considérez la modélisation d'une analyse (*assay*) d'une préparation (*batch*) chimique. La classe `Assay` possède un attribut `batch` de type `Batch`, et la classe `Batch` comprend un attribut `chemical` de type `Chemical`. Supposez qu'il y ait un certain objet `Assay` dont l'attribut `batch` se réfère à un objet `b` de type `Batch`, et aussi que l'attribut `chemical` de l'objet `b` se réfère à un objet `c` de type `Chemical`.

La Figure 5.3 illustre deux options de diagrammes possibles pour ce modèle. Pour plus d'informations sur l'illustration de modèles objet avec UML, voyez l'Annexe D.

Figure 5.3

*Deux options de représentation possible des mêmes informations :
l'objet a référence
l'objet b, et l'objet b
référence l'objet c.*

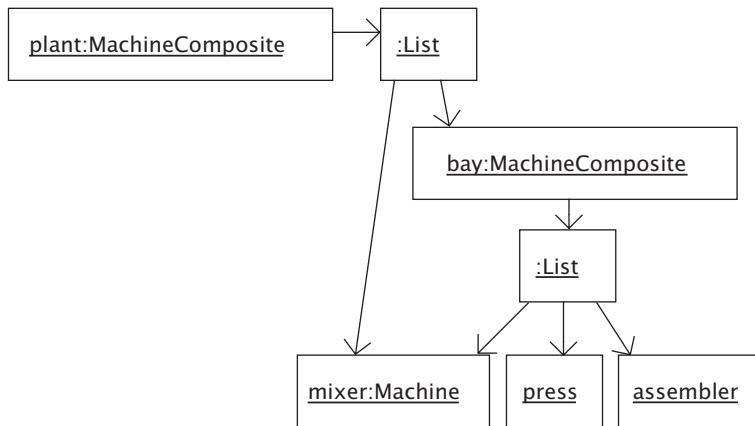


Il y a un **chemin**, une série de références d'objets, de a à c, car a référence b et b référence c. Un **cycle** est un chemin le long duquel un certain nœud apparaît deux fois. Il y aurait un cycle de références dans ce modèle si l'objet Chemical c référait en retour l'objet Assay a. Les modèles objet sont des **graphes orientés** car chaque référence d'objet possède une direction. La **théorie des graphes** applique généralement le terme **arbre** pour désigner certains graphes non orientés. Un graphe orienté peut toutefois être appelé un arbre si :

- Il possède un nœud **racine** qui n'est pas référencé.
- Chaque autre nœud n'a qu'un **parent**, le nœud qui le référence.

Pourquoi se préoccuper de cette notion d'arbre pour un graphe ? Parce que le pattern COMPOSITE convient particulièrement bien aux structures qui suivent cette forme — comme nous le verrons, vous pouvez néanmoins faire fonctionner un COMPOSITE avec un graphe orienté acyclique ou même un graphe cyclique, mais cela demande du travail et une attention supplémentaires.

Le modèle objet illustré à la Figure 5.3 est un simple arbre. Lorsque les modèles sont de plus grande taille, il peut être difficile de savoir s'il s'agit d'un arbre. La Figure 5.4 présente le modèle objet d'une usine, appelé **plant**, c'est-à-dire un objet **MachineComposite**. Cette usine comprend une travée composée de trois machines : un mixeur (**mixer**), une presse (**press**) et un assembleur (**assembler**). Le modèle montre aussi que la liste de composants-machines de l'objet **plant** contient une référence directe au mixeur.

**Figure 5.4**

Un modèle objet formant un graphe qui n'est ni cyclique, ni un arbre.

Le graphe d'objets de la Figure 5.4 ne comprend pas de cycle, mais ce n'est pas un arbre car deux objets référencent le même objet `mixer`. Si nous supprimons ou ne tenons pas compte de l'objet `plant` et de sa liste, l'objet `bay` est la racine de l'arbre.

Les méthodes qui s'appliquent à des composites peuvent avoir des défauts si elles supposent que tous les composites sont des arbres mais que le système accepte des composites qui n'en sont pas. L'Exercice 5.2 demandait la définition d'une opération `getMachineCount()`. L'implémentation de cette opération dans la classe `Machine`, telle que donnée dans la solution de l'exercice, est correcte :

```

public int getMachineCount() {
    return 1;
}
  
```

La classe `MachineComposite` implémente aussi correctement `getMachineCount()`, retournant la somme des comptes de chaque composant d'un composite :

```

public int getMachineCount() {
    int count = 0;
    Iterator i = components.iterator();
    while (i.hasNext()) {
        MachineComponent mc = (MachineComponent) i.next();
        count += mc.getMachineCount();
    }
    return count;
}
  
```

Ces méthodes sont correctes tant que les objets `MachineComponent` sont des arbres. Il peut toutefois arriver qu'un composite que vous supposiez être un arbre ne le soit soudain plus. Cela se produirait vraisemblablement si les utilisateurs pouvaient modifier la composition. Considérez un exemple susceptible de se produire chez Oozinoz.

Les ingénieurs d'Oozinoz utilisent une application avec GUI pour enregistrer et actualiser la composition du matériel dans l'usine. Un jour, ils signalent un défaut concernant le nombre de machines rapporté existant dans l'usine. Vous pouvez reproduire leur modèle objet avec la méthode `plant()` de la classe `OozinozFactory` :

```
public static MachineComposite plant() {
    MachineComposite plant = new MachineComposite(100);
    MachineComposite bay = new MachineComposite(101);
    Machine mixer = new Mixer(102);
    Machine press = new StarPress(103);
    Machine assembler = new ShellAssembler(104);
    bay.add(mixer);
    bay.add(press);
    bay.add(assembler);
    plant.add(mixer);
    plant.add(bay);
    return plant;
}
```

Ce code produit l'objet `plant` vu plus haut dans la Figure 5.4.

Exercice 5.4

Que renvoie en sortie le programme suivant ?

```
package app.composite;
import com.oozinoz.machine.*;

public class ShowPlant {
    public static void main(String[] args) {
        MachineComponent c = OozinozFactory.plant();
        System.out.println(
            "Nombre de machines : " + c.getMachineCount());
    }
}
```

L'application avec GUI utilisée chez Oozinoz pour construire les modèles objet de l'équipement d'une usine devrait vérifier si un nœud existe déjà dans un arbre de composant avant de l'ajouter une seconde fois. Un moyen d'accomplir cela est de conserver un ensemble des nœuds existants. Il peut toutefois arriver que vous n'ayez pas le contrôle sur la formation d'un composite. Dans ce cas, vous pouvez écrire une méthode `isTree()` pour vérifier si un composite est un arbre.

Nous considérerons un modèle objet comme étant un arbre si un algorithme peut parcourir ses références sans traverser deux fois le même noeud. Vous pouvez implémenter une méthode `isTree()` sur la classe abstraite `MachineComponent` afin de déléguer l'appel à une méthode `isTree()` conservant un ensemble des nœuds parcourus. La classe `MachineComponent` peut laisser abstraite l'implémentation de la méthode `isTree(set:Set)` paramétrée. La Figure 5.5 illustre le placement des méthodes `isTree()`.

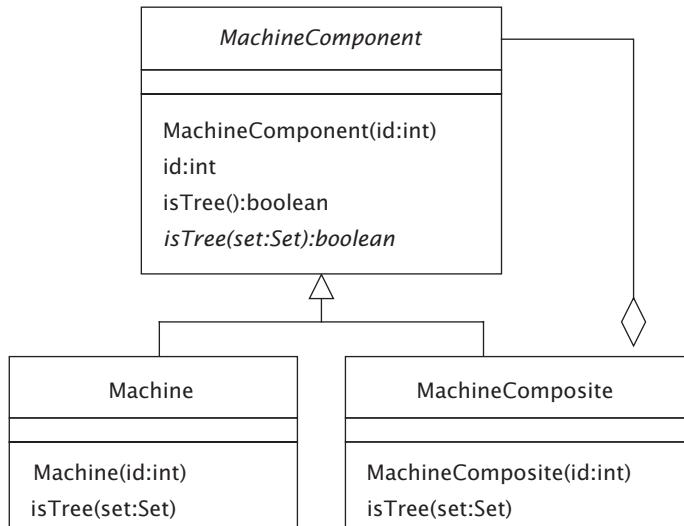
Le code de `MachineComponent` délègue un appel `isTree()` à sa méthode abstraite `isTree(s:Set)` :

```
public boolean isTree() {
    return isTree(new HashSet());
}
protected abstract boolean isTree(Set s);
```

Ces méthodes emploient la classe `Set` de la bibliothèque de classes Java.

Figure 5.5

Une méthode `isTree()` peut détecter si un composite est en réalité un arbre.



Les classes `Machine` et `MachineComposite` doivent implémenter la méthode abstraite `isTree(s:Set)`. L'implémentation de `isTree()` pour `Machine` est simple, reflétant le fait que des machines individuelles sont toujours des arbres :

```
protected boolean isTree(Set visited) {  
    visited.add(this);  
    return true;  
}
```

L'implémentation dans `MachineComposite` de `isTree()` doit ajouter l'objet récepteur à la collection `visited` puis parcourir tous les composants du composite. La méthode peut retourner `false` si un composant a déjà été parcouru ou n'est pas un arbre. Sinon, elle retourne `true`.

Exercice 5.5

Ecrivez le code pour `MachineComposite.isTree(Set visited)`.

En procédant avec soin, vous pouvez garantir qu'un modèle objet reste un arbre en refusant tout changement qui ferait retourner `false` par `isTree()`. D'un autre côté, vous pouvez décider d'autoriser l'existence de composites qui ne sont pas des arbres, surtout lorsque le domaine de problèmes que vous modélisez contient des cycles.

Des composites avec des cycles

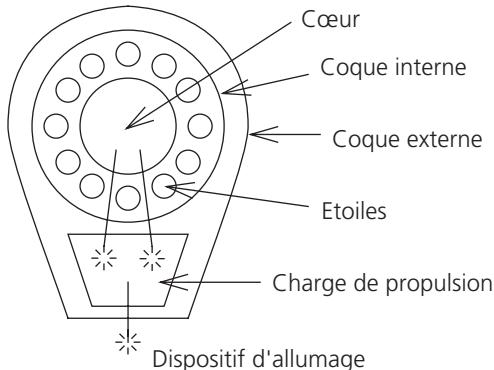
Le composite non-arbre auquel se référait l'Exercice 5.4 était un accident dû au fait qu'un utilisateur avait marqué une machine comme faisant partie à la fois d'une usine (*plant*) et d'une travée (*bay*). Pour les objets physiques, vous pouvez préférer interdire le concept d'objet contenu par plus d'un autre objet. Toutefois, un domaine de problèmes peut comprendre des éléments non physiques pour lesquels des cycles de confinement sont justifiés. Cela se produit fréquemment lors de la modélisation de flux de processus.

Considérez la construction de bombes aériennes telles que celle illustrée Figure 5.6. Une bombe est lancée au moyen d'un mortier, ou tube, par la mise à feu de la charge de propulsion (contenant de la poudre noire) logée sous la charge centrale. Le deuxième dispositif d'allumage brûle alors que la bombe est en l'air pour finalement atteindre la charge centrale lorsque la bombe est à son apogée.

Lorsque celle-ci explose, les étoiles mises à feu produisent les effets visuels des feux d'artifice.

Figure 5.6

Une bombe aérienne utilise deux charges : l'une pour la propulsion initiale et l'autre pour faire éclater le cœur contenant les étoiles lorsque la bombe atteint son apogée.



Le flux des processus de construction d'une bombe aérienne commence par la fabrication d'une coque interne, suivie d'une vérification, puis d'une amélioration ou de son assemblage final.

Pour fabriquer la coque interne, un opérateur utilise un assembleur de coques qui place les étoiles dans un compartiment hémisphérique, insère une charge centrale de poudre noire, ajoute davantage d'étoiles au-dessus de la charge, puis ferme le tout au moyen d'un autre compartiment hémisphérique.

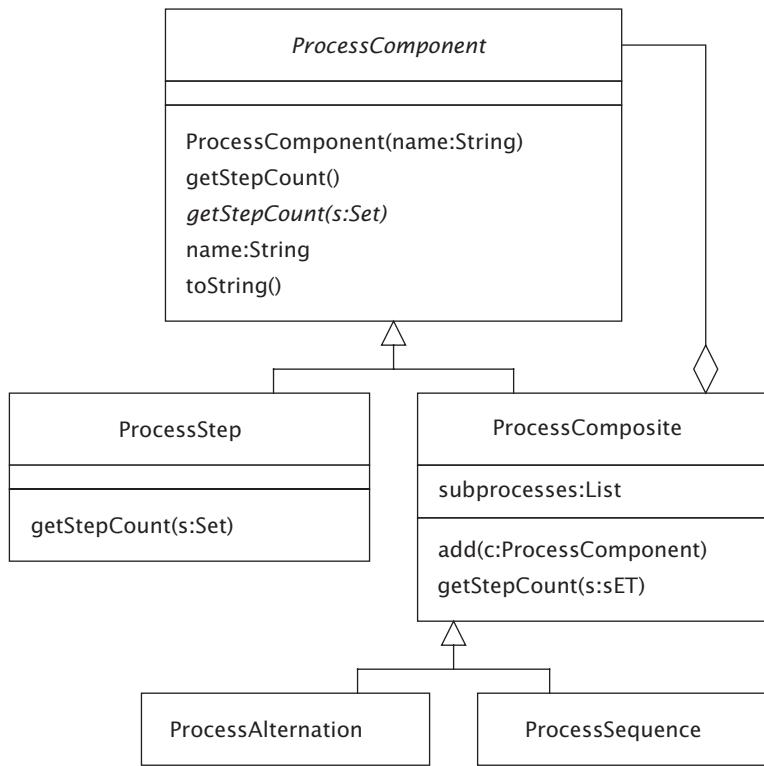
Un inspecteur vérifie que la coque interne répond aux standards de sécurité et de qualité. Si ce n'est pas le cas, l'opérateur la désassemble et recommence. Si elle passe l'inspection, l'opérateur ajoute un dispositif d'allumage pour joindre une charge de propulsion à la coque interne, puis termine en ajoutant une enveloppe.

Comme pour les composites de machines, les ingénieurs d'Oozinoz disposent d'une application avec GUI leur permettant de décrire la composition d'un processus. La Figure 5.7 montre la structure des classes qui gèrent la modélisation du processus.

La Figure 5.8 présente les objets qui représentent le flux des processus participant à la fabrication d'une bombe aérienne. Le processus `make` est une séquence composée de l'étape `buildInner` suivie de l'étape `inspect` et du sous-processus `reworkOrFinish`. Ce sous-processus prend l'une des deux voies possibles. Il peut requérir une étape de désassemblage suivie du processus `make`, ou seulement d'une étape `finish`.

Figure 5.7

Le processus de construction de pièces d'artifice inclut des étapes qui sont des alternances ou des séquences d'autres étapes.



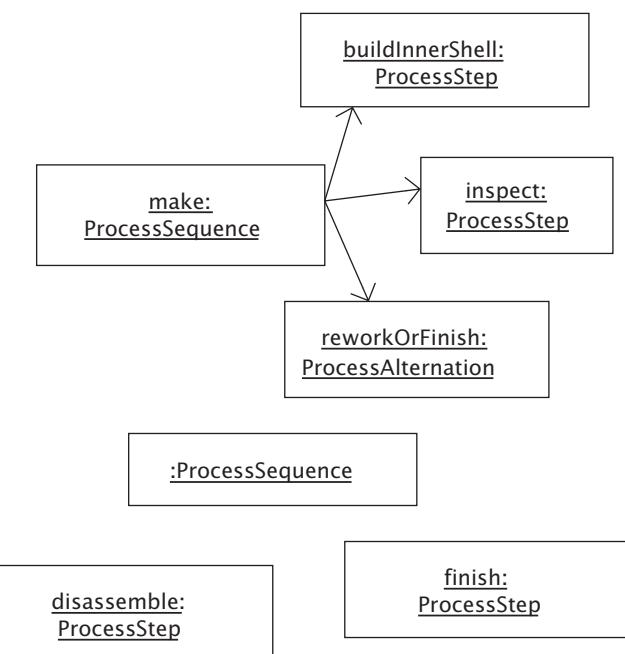
Exercice 5.6

La Figure 5.8 illustre les objets du modèle du processus d'assemblage d'une bombe. Un diagramme objet complet montrerait les relations entre tous les objets se référençant. Par exemple, le diagramme montre les références que l'objet `make` entretient. Votre travail est de compléter les relations manquantes dans le diagramme.

L'opération `getStepCount()` dans la hiérarchie `ProcessComponent` compte le nombre d'étapes individuelles dans le flux de processus. Notez que ce compte n'est pas la longueur du processus mais le nombre d'étapes de traitement de nœuds feuilles du graphe du processus. La méthode `getStepCount()` doit prendre soin de compter une fois chaque étape et de ne pas entrer dans une boucle infinie lorsqu'un processus comprend un cycle. La classe `ProcessComponent` implémente la

Figure 5.8

Une fois terminé, ce diagramme représentera un modèle objet du processus de fabrication de bombes aériennes à Oozinoz.



méthode `getStepCount()` de sorte qu'elle s'appuie sur une méthode compagnon qui transmet un ensemble de nœuds parcourus :

```

public int getStepCount() {
    return getStepCount(new HashSet());
}
public abstract int getStepCount(Set visited);
  
```

La classe `ProcessComposite` veille dans son implémentation à ce que la méthode `getStepCount()` ne parcourt pas un nœud déjà visité :

```

public int getStepCount(Set visited) {
    visited.add(getName());
    int count = 0;
    for (int i = 0; i < subprocesses.size(); i++) {
        ProcessComponent pc =
            (ProcessComponent) subprocesses.get(i);
        if (!visited.contains(pc.getName()))
            count += pc.getStepCount(visited);
    }
    return count;
}
  
```

L'implémentation de `getStepCount()` de la classe `ProcessStep` est simple :

```
public int getStepCount(Set visited) {
    visited.add(name);
    return 1;
}
```

Le package `com.oozinoz.process` d'Oozinoz contient une classe `ShellProcess` qui inclut une méthode `make()` qui retourne l'objet `make` illustré Figure 5.8. Le package `com.oozinoz.testing` comprend une classe `ProcessTest` qui fournit des tests automatisés de divers types de graphes de processus. Par exemple, la classe `ProcessTest` inclut une méthode qui vérifie que l'opération `getStepCount()` compte correctement le nombre d'étapes dans le processus `make` cyclique :

```
public void testShell() {
    assertEquals(4, ShellProcess.make().getStepCount());
}
```

Ce test s'exécute au sein du framework **JUnit**. Voir www.junit.org pour plus d'informations sur JUnit.

Conséquences des cycles

Beaucoup d'opérations sur un composite, telles que le calcul de son nombre de nœuds feuilles, sont justifiées même si le composite n'est pas un arbre. Généralement, la seule différence que les composites non-arbre introduisent est que vous devez être attentif à ne pas opérer une deuxième fois sur un même nœud. Toutefois, certaines opérations deviennent inutiles si le composite contient un cycle. Par exemple, nous ne pouvons pas déterminer par voie algorithmique le nombre maximal d'étapes requises pour fabriquer une bombe aérienne chez Oozinoz car le nombre de fois où l'étape d'amélioration doit être recommandée ne peut être connu. Toute opération dépendant de la longueur d'un chemin dans un composite ne serait pas logique si le composite comprend un cycle. Aussi, bien que nous puissions parler de la hauteur d'un arbre — le chemin le plus long de la racine à une feuille —, il n'y a pas de longueur de chemin maximale dans un graphe cyclique.

Une autre conséquence de permettre l'introduction de composites non-arbre est que vous perdez la capacité de supposer que chaque nœud n'a qu'un parent. Si un composite n'est pas un arbre, un nœud peut avoir plus d'un parent. Par exemple, le processus modélisé dans la Figure 5.8 pourrait avoir plusieurs étapes composées utilisant l'étape `inspect`, donnant ainsi à l'objet `inspect` plusieurs parents.

Il n'y a pas de problème inhérent au fait d'avoir un nœud avec plusieurs parents, mais votre modèle et votre code doivent en tenir compte.

Résumé

Le pattern COMPOSITE comprend deux concepts puissants associés. Le premier est qu'un groupe d'objets peut contenir des éléments individuels mais aussi d'autres groupes. L'autre idée est que ces éléments individuels et composites partagent une interface commune. Ces concepts sont unis dans la modélisation objet, lorsque vous créez une classe abstraite ou une interface Java qui définit des comportements communs à des objets composites et individuels.

La modélisation de composites conduit souvent à une définition récursive des méthodes sur les nœuds composites. Lorsqu'il y a une récursivité, il y a le risque d'écrire du code produisant une boucle infinie. Pour éviter ce problème, vous pouvez prendre des mesures pour garantir que vos composites soient toujours des arbres. Une autre possibilité est d'autoriser l'intervention de cycles dans un composite, mais il vous faut modifier vos algorithmes pour éviter toute récursivité infinie.

6

BRIDGE

Le pattern BRIDGE, ou Driver, vise à implémenter une abstraction. Le terme **abstraction** se réfère à une classe qui s'appuie sur un ensemble d'opérations abstraites, lesquelles peuvent avoir plusieurs implémentations.

La façon habituelle d'implémenter une abstraction est de créer une hiérarchie de classes, avec une classe abstraite au sommet qui définit les opérations abstraites. Chaque sous-classe de la hiérarchie apporte une implémentation différente de l'ensemble d'opérations. Cette approche devient insuffisante lorsqu'il vous faut dériver une sous-classe de la hiérarchie pour une quelconque autre raison.

Vous pouvez créer un BRIDGE (pont) en déplaçant l'ensemble d'opérations abstraites vers une interface de sorte qu'une abstraction dépendra d'une implémentation de l'interface.

L'objectif du pattern BRIDGE est de découpler une abstraction de l'implémentation de ses opérations abstraites, permettant ainsi à l'abstraction et à son implémentation de varier indépendamment.

Une abstraction ordinaire

Presque chaque classe est une abstraction dans ce sens qu'elle constitue une approximation, une idéalisation, ou une simplification de la catégorie d'objets réels qu'elle modélise. Toutefois, dans le cas du BRIDGE, nous utilisons spécifiquement le terme *abstraction* pour signifier une classe s'appuyant sur un ensemble d'opérations abstraites.

Supposez que vous ayez des classes de contrôle qui interagissent avec certaines machines produisant les pièces d'artifice chez Oozinoz. Ces classes reflètent les différences dans la façon dont les machines opèrent. Vous pourriez toutefois désirer créer certaines opérations abstraites qui produiraient les mêmes résultats sur n'importe quelle machine. La Figure 6.1 montre des classes de contrôle provenant du package `com.oozinoz.controller`.

Figure 6.1

Ces deux classes ont des méthodes semblables que vous pouvez placer dans un modèle commun pour piloter des machines.

<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 5px; text-align: center;">StarPressController</td></tr> <tr> <td style="padding: 10px; vertical-align: top;"> start() stop() startProcess() endProcess() index() discharge() </td></tr> </table>	StarPressController	start() stop() startProcess() endProcess() index() discharge()	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 5px; text-align: center;">FuserController</td></tr> <tr> <td style="padding: 10px; vertical-align: top;"> startMachine() stopMachine() begin() end() conveyIn() conveyOut() switchSpool() </td></tr> </table>	FuserController	startMachine() stopMachine() begin() end() conveyIn() conveyOut() switchSpool()
StarPressController					
start() stop() startProcess() endProcess() index() discharge()					
FuserController					
startMachine() stopMachine() begin() end() conveyIn() conveyOut() switchSpool()					

Les deux classes de la Figure 6.1 possèdent des méthodes semblables pour démarrer et arrêter les machines qu'elles contrôlent : presse à étoiles (*star press*) ou assembleuse de dispositif d'allumage (*fuser*). Elles sont toutefois nommées différemment : `start()` et `stop()` dans la classe `StarPressController`, et `startMachine()` et `stopMachine()` dans `FuserController`. Ces classes de contrôle, ou contrôleurs, possèdent également des méthodes pour amener une caisse dans la zone de traitement (`index()` et `conveyIn()`), pour débuter et terminer le traitement d'une caisse (`startProcess()` et `endProcess()`, et `begin()` et `end()`), et pour retirer une caisse (`discharge()` et `conveyOut()`). La classe `FuserController` possède également une méthode `switchSpool()` qui permet de changer la bobine de mèche d'allumage (*fuse spool*).

Supposez maintenant que vous souhaitez créer une méthode `shutdown()` qui assure un arrêt en bon ordre, effectuant les mêmes étapes sur les deux machines. Pour en simplifier l'écriture, vous pouvez standardiser les noms des opérations courantes, comme `startMachine()`, `stopMachine()`, `startProcess()`, `stopProcess()`, `conveyIn()`, et `conveyOut()`. Il se trouve toutefois que vous ne pouvez pas changer les classes de contrôle car l'une d'elles provient du fournisseur de la machine.

Exercice 6.1

Indiquez de quelle manière vous pourriez appliquer un pattern de conception pour permettre le contrôle de diverses machines avec une interface commune.

- Les solutions des exercices de ce chapitre sont données dans l'Annexe B.

La Figure 6.2 illustre l'introduction d'une classe abstraite `MachineManager` avec des sous-classes qui retransmettent les appels de contrôle en les adaptant au sein de méthodes supportées par `FuserController` et `StarPressController`.

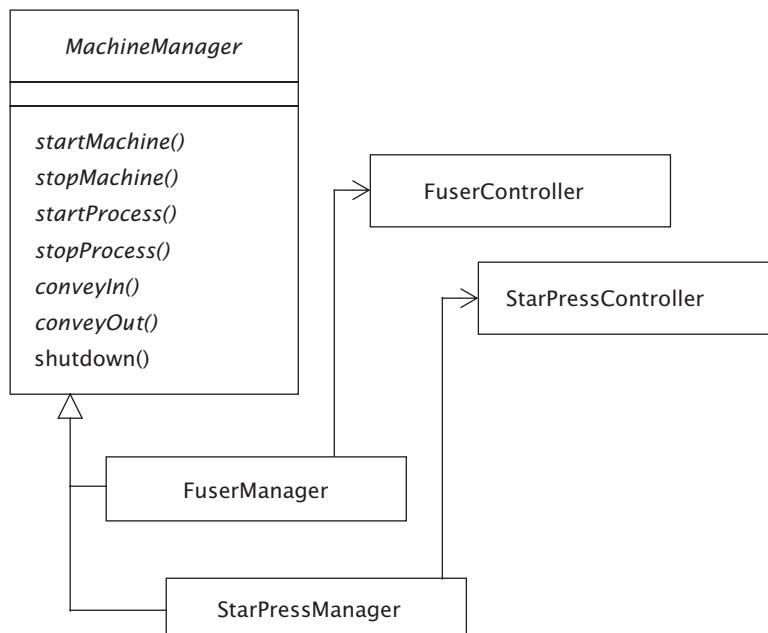


Figure 6.2

Les classes `FuserManager` et `StarPressManager` implémentent les méthodes abstraites de `MachineManager` en transmettant les appels aux méthodes correspondantes des objets `FuserController` et `StarPressController`.

Il n'est pas problématique qu'un contrôleur incorpore des opérations qui soient uniques pour le type de machine concerné. Par exemple, bien que la Figure 6.2 ne le montre pas, la classe `FuserManager` possède également une méthode

`switchSpool()` qui transmet les appels à la méthode `switchSpool()` d'un objet `FuserController`.

Exercice 6.2

Ecrivez une méthode `shutdown()` qui terminera le traitement pour la classe `MachineManager`, déchargera la caisse en cours de traitement et arrêtera la machine.

La méthode `shutdown()` de la classe `MachineManager` n'est pas abstraite mais concrète. Toutefois, nous pouvons dire que c'est une *abstraction* car la méthode universalise, ou abstrait, la définition des étapes à réaliser pour arrêter une machine.

De l'*abstraction* au pattern BRIDGE

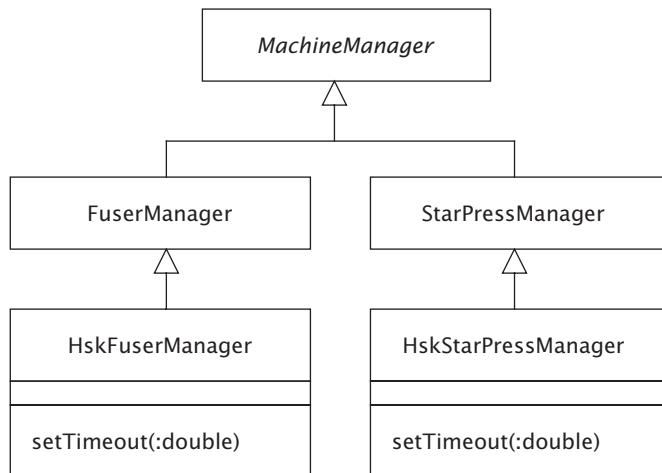
La hiérarchie `MachineManager` étant codée par rapport au type d'équipement, chaque type de machine nécessite une sous-classe différente de `MachineManager`. Que se passerait-il si vous deviez organiser la hiérarchie selon un autre critère ? Par exemple, supposez que vous travailliez directement sur les machines et que celles-ci fournissent un acquittement des étapes qu'elles accomplissent. Conformément à cela, vous voulez créer une sous-classe `MachineManager` de mise en route, ou *handshaking*, avec des méthodes permettant de paramétrier l'interaction avec la machine, telle que la définition d'une valeur de temporisation. Vous avez toutefois besoin de différents gestionnaires de machine pour les presses à étoiles et les assembleuses de dispositif d'allumage. Si vous ne réorganisiez pas d'abord la hiérarchie `MachineManager`, votre nouvelle hiérarchie risquerait de ressembler au modèle de la Figure 6.3.

La hiérarchie illustrée à la Figure 6.3 conçoit les classes suivant deux critères : selon le type de machine et selon que la machine gère ou non le protocole de mise en route. Ce principe dual de codage présente un problème. Plus particulièrement, une méthode telle que `setTimeout()` peut contenir un code identique à deux endroits, mais nous ne pouvons pas le coder dans la hiérarchie car les super-classes ne gèrent pas l'idée du handshaking.

En général, les classes de handshaking ne disposent d'aucun moyen pour partager le code car il n'y a pas de super-classe de handshaking. Et à mesure que nous ajoutons

Figure 6.3

Les sous-classes de mise en route (Hsk) ajoutent un paramétrage pour le temps d'attente d'un acquittement de la part d'une machine.



davantage de classes dans la hiérarchie, le problème empire. Si nous disposons au final de contrôleurs pour cinq machines et que la méthode `setTimeout()` doive être changée, nous devons modifier le code à cinq endroits.

Dans une telle situation, nous pouvons appliquer le pattern BRIDGE. Nous pouvons dissocier l'abstraction `MachineManager` de l'implémentation de ses opérations abstraites en plaçant les méthodes abstraites dans une hiérarchie distincte. La classe `MachineManager` demeure une abstraction et le résultat produit par l'appel de ses méthodes sera différent selon qu'il s'agira d'une presse ou d'une assemblageuse.

Séparer l'abstraction de l'implémentation de ses méthodes permet aux deux hiérarchies de varier de manière indépendante. Nous pouvons ajouter un support pour de nouvelles machines sans influer sur la hiérarchie `MachineManager`. Nous pouvons également étendre la hiérarchie `MachineManager` sans changer aucun des contrôleurs de machine. La Figure 6.4 présente la séparation souhaitée.

L'objectif de la nouvelle conception est de séparer la hiérarchie `MachineManager` de l'implémentation des opérations abstraites de la hiérarchie.

Exercice 6.3

La Figure 6.4 illustre la hiérarchie `MachineManager` restructurée en BRIDGE. Ajoutez les mentions manquantes.

Notez que dans la Figure 6.4 la classe `MachineManager2` devient concrète bien qu'elle soit toujours une abstraction. Les méthodes abstraites dont dépend maintenant `MachineManager` résident dans l'interface `MachineDriver`. Le nom de cette interface suggère que les classes qui adaptent les requêtes de `MachineManager` aux différentes machines spécifiques sont devenues des drivers. Un **driver** est un objet qui pilote un système informatique ou un équipement externe selon une interface bien spécifiée. Les drivers fournissent l'exemple le plus courant d'application du pattern BRIDGE.

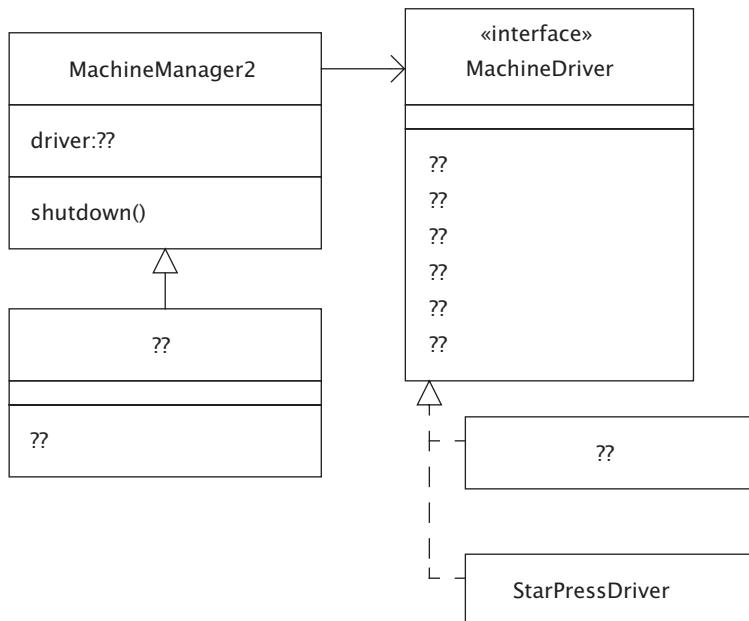


Figure 6.4

Une fois complété, ce diagramme montrera la séparation de l'abstraction MachineManager de l'implémentation de ses opérations abstraites.

Des drivers en tant que BRIDGE

Les drivers sont des abstractions. Le résultat de l'exécution de l'application dépend du driver en place. Chaque driver est une instance du pattern ADAPTER, fournissant l'interface qu'un client attend en utilisant les services d'une classe comportant une interface différente. Une conception globale qui utilise des drivers est une instance

de BRIDGE. La conception sépare le développement d'application de celui des drivers qui implémentent les opérations abstraites dont dépendent les applications.

Une conception à base de drivers vous force à créer un modèle abstrait commun de la machine ou du système à piloter. Cela présente l'avantage de permettre au code du côté abstraction de s'appliquer à n'importe lequel des drivers au travers desquels il pourrait s'exécuter. La définition d'un ensemble commun de méthodes pour les drivers peut toutefois présenter l'inconvénient d'éliminer le comportement qu'un équipement piloté pourrait supporter. Rappelez-vous de la Figure 6.1 qu'un contrôleur d'assemblage possède une méthode `switchSpool()`. Où cette méthode est-elle passée dans la conception révisée de la Figure 6.4 (ou Figure B5 de l'Annexe B) ? La réponse est que nous l'avons éliminée par abstraction. Vous pouvez l'inclure dans la nouvelle classe `FuserDriver`. Toutefois, ceci peut donner lieu à du code côté abstraction devant procéder à une vérification pour savoir si son driver est une instance de `FuserDriver`.

Pour éviter de perdre la méthode `switchSpool()`, nous pourrions faire en sorte que chaque driver l'implémente, sachant que certains d'entre eux ignoreront simplement l'appel. Lorsque vous devez choisir un modèle abstrait des opérations qu'un driver doit gérer, vous êtes souvent confronté à ce genre de décision. Vous pouvez inclure des méthodes que certains drivers ne supporteront pas, ou exclure des méthodes pour limiter ce que les abstractions pourront faire avec un driver ou bien les forcer à inclure du code pour un cas particulier.

Drivers de base de données

Un exemple banal d'application utilisant des drivers est l'accès à une base de données. La connectivité base de données dans Java s'appuie sur **JDBC**. Une bonne source de documentation expliquant comment appliquer JDBC est *JDBC™ API Tutorial and Reference (2/e)* [White et al. 1999]. Dit succinctement, JDBC est une API (*Application Programming Interface*) qui permet d'exécuter des instructions SQL (*Structured Query Langage*). Les classes qui implémentent l'interface sont des drivers JDBC, et les applications qui s'appuient sur ces drivers sont des abstractions qui peuvent fonctionner avec n'importe quelle base de données pour laquelle il existe un driver JDBC. L'architecture JDBC dissocie une abstraction de son implémentation pour que les deux puissent varier de manière indépendante ; c'est un excellent exemple de BRIDGE.

Pour utiliser un driver JDBC, vous le chargez, le connectez à la base de données et créez un objet Statement :

```
Class.forName(driverName);
Connection c = DriverManager.getConnection(url, user, pwd);
Statement stmt = c.createStatement();
```

Une description du fonctionnement de la classe DriverManager sortirait du cadre de la présente description. Sachez toutefois qu'à ce stade stmt est un objet Statement capable d'émettre des requêtes SQL qui retournent des ensembles de résultats (*result set*) :

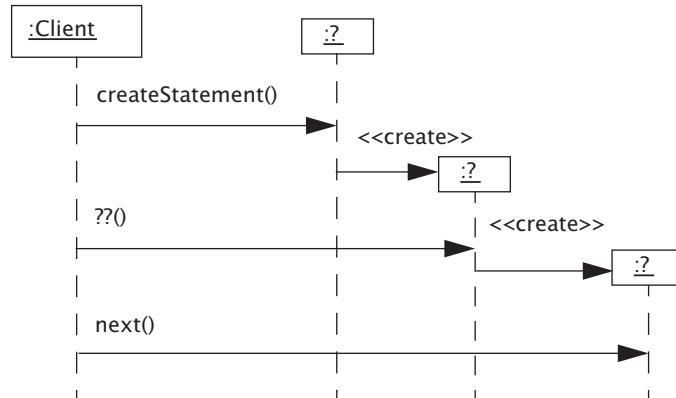
```
ResultSet result = stmt.executeQuery(
    "SELECT name, apogee FROM firework");
while (result.next()) {
    String name = result.getString("name");
    int apogee = result.getInt("apogee");
    System.out.println(name + ", " + apogee);
}
```

Exercice 6.4

La Figure 6.5 illustre un diagramme de séquence UML qui décrit le flux de messages dans une application JDBC typique. Complétez les noms de types et le nom de message manquants.

Figure 6.5

Ce diagramme montre une partie du flux de messages typique qui intervient dans une application JDBC.



Exercice 6.5

Supposez que chez Oozinoz nous n’ayons que des bases de données SQL Server. Donnez un argument en faveur de l’emploi de lecteurs et d’adaptateurs spécifiques à SQL Server. Donnez un autre argument qui justifierait de *ne pas* le faire.

L’architecture JDBC divise clairement les rôles du développeur de driver et du développeur d’application. Dans certains cas, cette division n’existera pas à l’avance, même si vous utilisez des drivers. Vous pourrez éventuellement implémenter des drivers en tant que sous-classes d’une super-classe abstraite, avec chaque sous-classe pilotant un sous-système différent. Dans une telle situation, vous pouvez envisager l’implémentation d’un BRIDGE lorsqu’il vous faut davantage de souplesse.

Résumé

Une abstraction est une classe qui dépend de méthodes abstraites. L’exemple le plus simple d’abstraction est une hiérarchie abstraite, où des méthodes concrètes dans la super-classe dépendent d’autres méthodes abstraites. Vous pouvez être forcé de déplacer ces dernières vers une autre hiérarchie si vous voulez restructurer la hiérarchie originale selon un autre critère. Il s’agit alors d’une application du pattern BRIDGE, séparant une abstraction de l’implémentation de ses méthodes abstraites.

L’exemple le plus courant d’application de BRIDGE apparaît dans les drivers, tels que ceux de base de données. Les drivers de base de données sont un bon exemple des compromis inhérents à une conception avec BRIDGE. Un driver peut nécessiter des méthodes qu’un implémenteur ne peut gérer. D’un autre côté, un driver peut négliger des méthodes utiles qui pourraient s’appliquer à une certaine base de données. Cela pourrait vous inciter à récrire du code spécifique à une implémentation au lieu d’être abstrait. Il n’est pas toujours évident de savoir s’il faut privilégier l’abstraction ou la spécificité, mais il est important de prendre des décisions mûrement réfléchies.

II

Patterns de responsabilité

Introduction à la responsabilité

La responsabilité d'un objet est comparable à celle d'un représentant au centre de réception des appels de la société Oozinoz. Lorsqu'un client appelle chez Oozinoz, la personne qui répond est un intermédiaire, ou *proxy* pour reprendre un terme informatique, qui représente la société. Ce représentant effectue des tâches prévisibles, généralement en les délégant à un autre système ou à une autre personne. Parfois, le représentant délègue une requête à une seule autorité centrale qui joue le rôle de médiateur dans une situation ou transmet les problèmes le long d'une chaîne de responsabilités.

A l'instar des représentants, les objets ordinaires disposent des informations et des méthodes adéquates pour pouvoir opérer de manière indépendante. Cependant, il y a des situations qui demandent de s'écartez de ce modèle de fonctionnement indépendant et de recourir à une entité centrale responsable. Il existe plusieurs patterns qui répondent à ce type de besoin. Il y a aussi des patterns qui permettent aux objets de relayer les requêtes et qui isolent un objet des autres objets qui en dépendent. Les patterns afférents à la responsabilité fournissent des techniques pour centraliser, transmettre et aussi limiter la responsabilité des objets.

Responsabilité ordinaire

Bien que vous ayez probablement une bonne idée de la façon dont les attributs et les responsabilités doivent être associés dans une classe bien conçue, il pourrait vous paraître difficile d'expliquer les raisons motivant vos choix.

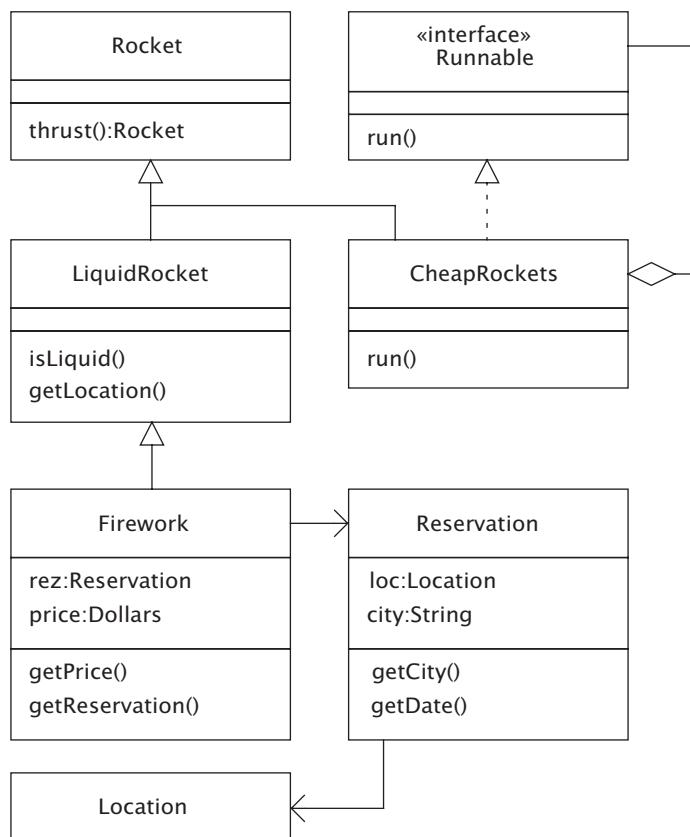
Exercice 7.1

La structure de la classe illustrée Figure 7.1 présente au moins dix choix d'assignation de responsabilités discutables. Identifiez tous les problèmes possibles et expliquez par écrit ce qui est erroné pour quatre d'entre eux.

- Les solutions des exercices de ce chapitre sont données dans l'Annexe B.

Figure 7.1

Qu'est-ce qui
ne va pas dans
cette figure ?



L'examen des bizarries de la Figure 7.1 débridera votre réflexion pour entreprendre une modélisation d'objets appropriée. C'est l'état d'esprit qui convient lorsque vous définissez des termes tels que *classe*. La valeur de l'exercice de définition de

termes augmente s'il favorise la communication entre les personnes et diminue s'il devient un objectif en soi et une source de conflits. Dans ce même état d'esprit, répondez à la difficile question de l'Exercice 7.2.

Exercice 7.2

Définissez les qualités d'une classe efficace et utile.

L'emploi d'une classe est facilité si ses méthodes ont un nom suffisamment explicite pour qu'on puisse savoir ce qu'elles réalisent. Il y a toutefois des cas où un nom de méthode ne contient pas suffisamment d'informations pour qu'on puisse prédire l'effet qu'aura un appel de la méthode.

Exercice 7.3

Donnez un exemple de raison pour laquelle l'impossibilité de prédire l'effet produit par un appel de méthode serait justifiée.

L'élaboration de principes relatifs à l'assignation de responsabilités dans un système orienté objet est un domaine qui nécessite des progrès. Un système dans lequel chaque classe et méthode définit clairement ses responsabilités et s'en acquitte correctement est un système puissant, supérieur à la plupart des systèmes que l'on rencontre aujourd'hui.

Contrôle de la responsabilité grâce à la visibilité

Il est courant de parler de classes et de méthodes assumant diverses responsabilités. Dans la pratique, cela signifie généralement que *vous*, en tant que développeur, assumez la responsabilité d'une conception robuste et d'un fonctionnement correct du code. Heureusement que Java apporte quelque soulagement dans ce domaine. Vous pouvez limiter la visibilité de vos classes, champs et méthodes, et circonscrire ainsi la responsabilité au niveau des développeurs qui emploient votre code. La visibilité peut être un signe de la façon dont une portion d'une classe doit être exposée.

Le Tableau 7.1 reprend les définitions informelles de l'effet des modificateurs d'accès.

Tableau 7.1 : Définitions informelles de l'effet des modificateurs d'accès

<i>Accès</i>	<i>Définition informelle</i>
public	Accès non limité
(rien)	Accès limité au package
protected	Accès limité à la classe contenant l'élément en question, ou aux types dérivés de cette classe
private	Accès limité au type contenant l'élément en question

Dans la pratique, certaines subtilités demandent de considérer plutôt la définition formelle de ces modificateurs d'accès qu'une définition intuitive. Par exemple, la question de savoir si une visibilité affecte des objets ou des classes.

Exercice 7.4

Un objet peut-il se référer à un membre privé d'une autre instance de la même classe ? Plus spécifiquement, le code suivant compilera-t-il ?

```
public class Firework {  
    private double weight = 0;  
    // ...  
    private double compare(Firework f) {  
        return weight - f.weight;  
    }  
}
```

Les modificateurs d'accès vous aident à limiter votre responsabilité en restreignant les services que vous fournissez aux autres développeurs. Par exemple, si vous ne voulez pas que d'autres développeurs puissent manipuler un champ de l'une de vos classes, vous pouvez le définir `private`. Inversement, vous apporterez de la souplesse pour les autres développeurs en déclarant un élément `protected`, bien qu'il y ait le risque d'associer trop étroitement les sous-classes à la classe parent. Prenez des décisions mûrement réfléchies et, au besoin, mettez en place une stratégie de groupe concernant la façon dont vous voulez restreindre les accès pour limiter vos responsabilités tout en autorisant des extensions futures.

Résumé

En tant que développeur Java, vous êtes responsable de la création des classes formant un ensemble logique d'attributs et de comportements associés. Créer une classe efficace est un art, mais il est possible d'établir certaines caractéristiques d'une classe bien conçue. Une de vos responsabilités est aussi de vous assurer que les méthodes de vos classes assurent bien les services que leur nom suggère. Vous pouvez limiter cette responsabilité avec l'emploi approprié de modificateurs de visibilité, mais envisagez l'existence de certains compromis entre sécurité et souplesse au niveau de la visibilité de votre code.

Au-delà de la responsabilité ordinaire

Indépendamment de la façon dont une classe restreint l'accès à ses membres, le développement OO répartit normalement les responsabilités entre objets individuels. En d'autres termes, le développement OO promeut l'**encapsulation**, l'idée qu'un objet travaille sur ses propres données.

La responsabilité distribuée est la norme, mais plusieurs patterns de conception s'y opposent et placent la responsabilité au niveau d'un objet intermédiaire ou d'un objet central. Par exemple, le pattern **SINGLETON** concentre la responsabilité au niveau d'un seul objet et fournit un accès global à cet objet. Une façon de se souvenir de l'objectif de ce pattern, ainsi que de celui d'autres patterns, est de les voir en tant qu'exceptions à la règle ordinaire de la responsabilité répartie.

<i>Si vous envisagez de</i>	<i>Appliquez le pattern</i>
• Centraliser la responsabilité au niveau d'une instance de classe	SINGLETON
• Libérer un objet de la "conscience" de connaître les objets qui en dépendent	OBSERVER
• Centraliser la responsabilité au niveau d'une classe qui supervise la façon dont les objets interagissent	MEDIATOR
• Laisser un objet agir au nom d'un autre objet	PROXY
• Autoriser une requête à être transmise le long d'une chaîne d'objets jusqu'à celui qui la traitera	CHAIN OF RESPONSABILITY
• Centraliser la responsabilité au niveau d'objets partagés de forte granularité	FLYWEIGHT

L'objectif de chaque pattern de conception est de permettre la résolution d'un problème dans un certain contexte. Les patterns de responsabilité conviennent dans des contextes où vous devez vous écarter de la règle normale stipulant que la responsabilité devrait être distribuée autant que possible.

8

SINGLETON

Les objets peuvent généralement agir de façon responsable en effectuant leur travail sur leurs propres attributs, sans avoir d'autre obligation que d'assurer leur cohérence propre. Cependant, certains objets assument d'autres responsabilités, telles que la modélisation d'entités du monde réel, la coordination de tâches, ou la modélisation de l'état général d'un système. Lorsque, dans un système, un certain objet assume une responsabilité dont dépendent d'autres objets, vous devez disposer d'une méthode pour localiser cet objet. Par exemple, vous pouvez avoir besoin d'identifier un objet qui représente une machine particulière, un objet client qui puisse se construire lui-même à partir d'informations extraites d'une base de données, ou encore un objet qui initie une récupération de la mémoire système.

Dans certains cas, lorsque vous devez trouver un objet responsable, l'objet dont vous avez besoin sera la seule instance de sa classe. Par exemple, la création de fusées peut se suffire d'un seul objet Factory. Dans ce cas, vous pouvez utiliser le pattern SINGLETON.

L'objectif du pattern SINGLETON est de garantir qu'une classe ne possède qu'une seule instance et de fournir un point d'accès global à celle-ci.

Le mécanisme de SINGLETON

Le mécanisme de SINGLETON est plus simple à exposer que son objectif. En effet, il est plus aisément d'expliquer *comment* garantir qu'une classe n'aura qu'une instance que *pourquoi* cette restriction est souhaitable. Vous pouvez placer SINGLETON dans la catégorie "patterns de création", comme le fait l'ouvrage *Design Patterns*.

L'essentiel est de voir les patterns d'une façon qui permette de s'en souvenir, de les reconnaître et de les appliquer. L'*intention*, ou l'objectif, du pattern **SINGLETON** demande qu'un objet spécifique assume une responsabilité dont dépendent d'autres objets.

Vous disposez de quelques options quant à la façon de créer un objet qui remplit un rôle de manière unique. Indépendamment de la façon dont vous créez un singleton, vous devez vous assurer que d'autres développeurs ne créent pas de nouvelles instances de la classe que vous souhaitez restreindre.

Exercice 8.1

Comment pouvez-vous empêcher d'autres développeurs de créer de nouvelles instances de votre classe ?

- Les solutions des exercices de ce chapitre sont données dans l'Annexe B.

Lorsque vous concevez une classe singleton, vous devez décider du moment de l'instanciation du seul objet qui représentera la classe. Une possibilité est de créer cette instance en tant que champ statique dans la classe. Par exemple, une classe **SystemStartup** pourrait inclure la ligne :

```
private static Factory factory = new Factory();
```

Cette classe pourrait rendre son unique instance disponible par l'intermédiaire d'une méthode **getFactory()** publique et statique.

Plutôt que de créer à l'avance une instance de singleton, vous pouvez attendre jusqu'au moment où l'instance est requise pour la première fois en utilisant une initialisation tardive, dite "paresseuse", ou *lazy-initialization*. Par exemple, la classe **SystemStartup** peut mettre à disposition son unique instance de la manière suivante :

```
public static Factory getFactory() {  
    if (factory == null)  
        factory = new Factory();  
    // ...  
    return factory;  
}
```

Exercice 8.2

Pour quelle raison décideriez-vous de procéder à l'initialisation paresseuse d'une instance de singleton plutôt que de l'initialiser dans la déclaration de son champ ?

Quoi qu'il en soit, le pattern SINGLETON suggère de fournir une méthode publique et statique qui donne accès à l'objet SINGLETON. Si cette méthode crée l'objet, elle doit aussi s'assurer que seule une instance pourra être créée.

Singletons et threads

Si vous voulez effectuer l'initialisation paresseuse d'un singleton dans un environnement multithread, vous devez prendre soin d'empêcher plusieurs threads d'initialiser le singleton. Dans une telle situation, il n'y a aucune garantie qu'une méthode se termine avant qu'une autre méthode dans un autre thread commence son exécution. Il se pourrait, par exemple, que deux threads tentent d'initialiser un singleton pratiquement en même temps. Supposez qu'une méthode détecte le singleton comme étant null. Si un autre thread débute à ce moment-là, il trouvera également le singleton à null. Les deux méthodes voudront alors l'initialiser. Pour empêcher ce type de contention, vous devez recourir à un mécanisme de verrouillage pour coordonner les méthodes s'exécutant dans différents threads.

Java inclut des fonctionnalités pour supporter le développement multithread, et, plus spécifiquement, il fournit à chaque objet un verrou (**lock**), une ressource exclusive qui indique que l'objet appartient à un thread. Pour garantir qu'un seul objet initialise un singleton, vous pouvez synchroniser l'initialisation par rapport au verrou d'un objet approprié. D'autres méthodes nécessitant l'accès exclusif au singleton peuvent être synchronisées par rapport au même verrou. Pour plus d'informations sur la POO avec concurrence, reportez-vous à l'excellent ouvrage *Concurrent Programming in Java™* [Lea 2000]. Ce livre suggère une synchronisation sur le verrou qui appartient à la classe elle-même, comme dans le code suivant :

```
package com.oozinoz.businessCore;  
import java.util.*;  
  
public class Factory {  
    private static Factory factory;
```

```
private static Object classLock = Factory.class;

private long wipMoves;

private Factory() {
    wipMoves = 0;
}
public static Factory getFactory() {
    synchronized (classLock) {
        if (factory == null)
            factory = new Factory();

        return factory;
    }
}
public void recordWipMove() {
    // Exercice !
}
```

Le code de `getFactory()` s'assure que si un second thread tente une initialisation paresseuse du singleton après qu'un autre thread a commencé la même initialisation, le second thread devra attendre pour obtenir le verrou d'objet `classLock`. Une fois qu'il obtiendra le verrou, il ne détectera pas de singleton `null` (étant donné qu'il ne peut y avoir qu'une seule instance de la classe, nous pouvons utiliser le seul verrou statique).

La variable `wipMoves` enregistre le nombre de fois que le travail en cours (*wip, work in process*), progresse. Chaque fois qu'une caisse arrive sur une autre machine, le sous-système qui provoque ou enregistre l'avancement doit appeler la méthode `recordWipMove()` du singleton `factory`.

Exercice 8.3

Ecrivez le code de la méthode `recordWipMove()` de la classe `Factory`.

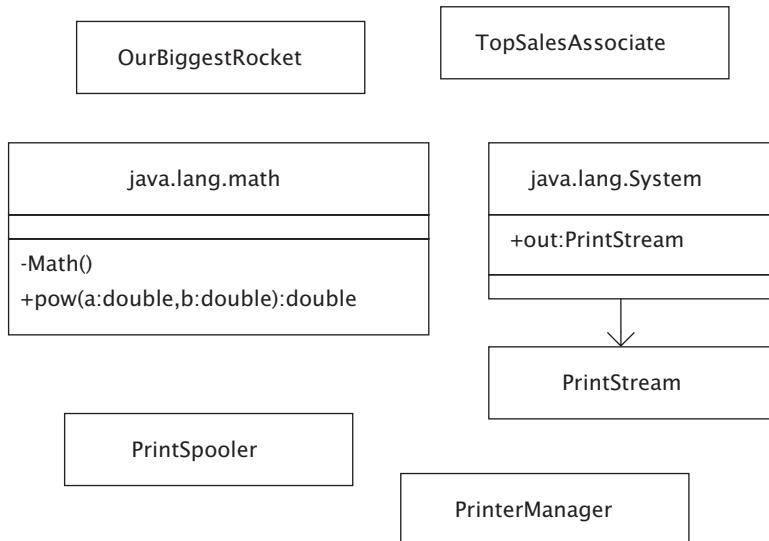
Identification de singltons

Les objets uniques ne sont pas chose inhabituelle. En fait, la plupart des objets dans une application assument une responsabilité unique. Pourquoi créer deux objets avec des responsabilités identiques ? De même, pratiquement chaque classe assure un rôle unique. Pourquoi développer deux fois la même classe ? D'un

autre côté, une classe singleton — une classe qui n'autorise qu'une instance — est relativement rare. Le fait qu'un objet ou une classe soit unique ne signifie pas nécessairement que le pattern SINGLETON est appliqué. Considérez les classes de la Figure 8.1.

Figure 8.1

Quelles classes semblent appliquer SINGLETON ?



Exercice 8.4

Pour chaque classe de la Figure 8.1, indiquez s'il s'agit d'une classe singleton et justifiez votre réponse.

SINGLETON est probablement le pattern le plus connu, mais il faut y recourir avec prudence car il est facile de mal l'utiliser. Ne laissez pas cette technique devenir un moyen pratique de créer des variables globales. Le couplage introduit n'est pas beaucoup mieux simplement parce que vous avez utilisé un pattern. Minimisez le nombre de classes qui savent qu'elles travaillent avec un SINGLETON. Il est préférable pour une classe de simplement savoir qu'elle a un objet avec lequel travailler et non de connaître les restrictions relatives à sa création. Soyez attentif à l'emploi

que vous souhaitez en faire. Par exemple, si vous avez besoin de sous-classes ou de différentes versions pour effectuer des tests, **SINGLETON** ne sera probablement pas approprié car il n'y aura pas exactement une seule instance.

Résumé

Le code qui supporte **SINGLETON** s'assure qu'une classe ne possède qu'une instance et fournit un point d'accès global à l'instance. Une façon de l'implémenter est de procéder par initialisation paresseuse d'un objet singleton, en l'instanciant seulement lorsqu'il est requis. Dans un environnement multithread, vous devez veiller à gérer la collaboration des threads qui peuvent accéder simultanément aux méthodes et aux données d'un singleton.

Le fait qu'un objet soit unique n'indique pas forcément que le pattern **SINGLETON** a été utilisé. Celui-ci centralise l'autorité au niveau d'une seule instance de classe en dissimulant le constructeur et en offrant un seul point d'accès à la méthode de création de l'objet.

OBSERVER

D'ordinaire, les clients collectent des informations provenant d'un objet en appelant ses méthodes. Toutefois, lorsque l'objet change, un problème survient : comment les clients qui dépendent des informations de l'objet peuvent-ils déterminer que celles-ci ont changé ?

Certaines conceptions imputent à l'objet la responsabilité d'informer les clients lorsqu'un aspect intéressant de l'objet change. Cette démarche pose un problème, car c'est le client qui sait quels sont les attributs qui l'intéressent. L'objet intéressant ne doit pas accepter la responsabilité d'actualiser le client. Une solution possible est de s'arranger pour que le client soit informé lorsque l'objet change et de laisser au client la liberté de s'enquérir ou non du nouvel état de l'objet.

L'objectif du pattern OBSERVER est de définir une dépendance du type un-à-plusieurs (1,n) entre des objets de manière que, lorsqu'un objet change d'état, tous les objets dépendants en soient notifiés afin de pouvoir réagir conformément.

Un exemple classique : OBSERVER dans les interfaces utilisateurs

Le pattern OBSERVER permet à un objet de demander d'être notifié lorsqu'un autre objet change. L'exemple le plus courant d'application de ce pattern intervient dans les interfaces graphiques utilisateurs, ou GUI. Lorsqu'un utilisateur clique sur un bouton ou agit sur un curseur, de nombreux objets de l'application doivent réagir au changement. Les concepteurs de Java ont anticipé l'intérêt de savoir quand un utilisateur provoque un changement au niveau d'un composant de l'interface graphique.

La forte présence de `OBSERVER` dans Swing nous le prouve. Swing se réfère aux clients en tant que *listeners* et vous pouvez enregistrer autant de listeners que vous le souhaitez pour recevoir les événements d'un objet.

Considérez une application typique Oozinoz avec GUI, telle celle illustrée Figure 9.1. Cette application permet à un ingénieur de tester visuellement les paramètres déterminant la relation entre la poussée (*thrust*), le taux de combustion (*burn rate*) et la surface de combustion (*burn surface*).

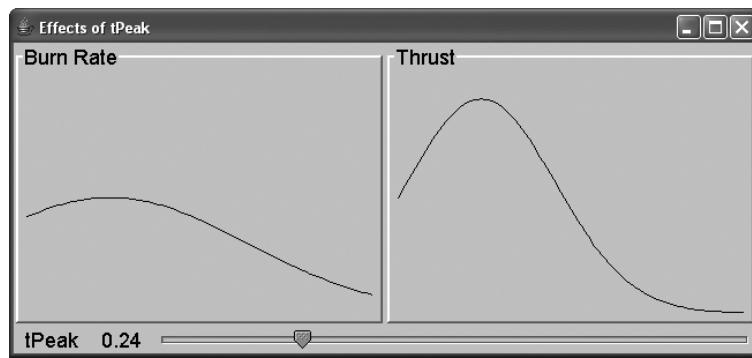


Figure 9.1

Les courbes montrent le changement en temps réel lorsque l'utilisateur ajuste la variable t_{peak} avec le curseur.

Lors de la mise à feu d'une fusée, la portion de combustible qui est exposée à l'air brûle et provoque une poussée. De la mise à feu au taux de combustion maximal, la surface de combustion, partie de la zone initiale d'allumage, augmente pour atteindre la surface totale du combustible. Ce taux maximal se produit au moment t_{peak} . La surface de combustion diminue ensuite à mesure que le combustible est consommé. L'application de calculs balistiques normalise le temps pour qu'il soit à 0 lors de l'allumage et à 1 lorsque la combustion s'arrête. Aussi, t_{peak} représente un nombre entre 0 et 1.

Oozinoz utilise un jeu d'équations de calcul du taux de combustion et de la poussée :

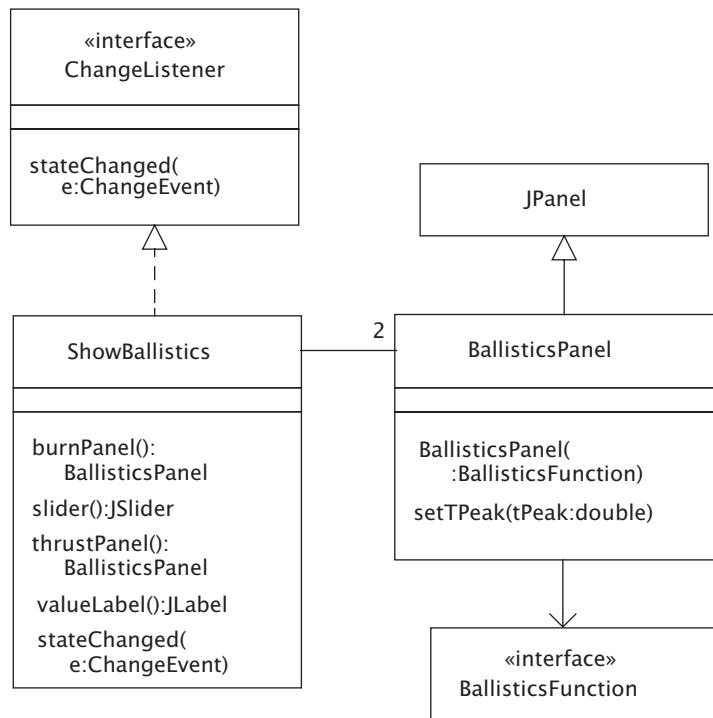
$$taux = 25^{-(t - t_{peak})^2}$$

$$poussée = 1,7 \left(\frac{taux}{0,6} \right)^{1/0,3}$$

L’application représentée Figure 9.1 montre comment t_{peak} influe sur le taux de combustion et la poussée d’une fusée. A mesure que l’utilisateur déplace le curseur, la valeur de t_{peak} change et les courbes suivent une autre forme. La Figure 9.2 illustre les classes principales constituant l’application.

Figure 9.2

L’application de calculs balistiques s’enregistre elle-même pour recevoir les événements de curseur.



Les classes `ShowBallistics` et `BallisticsPanel` sont des membres du package `app.observer.ballistics`. L’interface `BallisticsFunction` est un membre du package `com.oozinoz.ballistics`. Ce package contient aussi une classe utilitaire `Ballistics` fournissant les instances de `BallisticsFunction` qui définissent les courbes pour le taux de combustion et la poussée.

Lorsque l’application initialise le curseur, elle s’enregistre elle-même pour en recevoir les événements. Lorsque la valeur du curseur change, l’application actualise les panneaux d’affichage (*Panel*) des courbes ainsi que l’étiquette (*Label*) affichant la valeur de t_{peak} .

Exercice 9.1

Complétez les méthodes `slider()` et `stateChanged()` pour `ShowBallistics` de manière que les panneaux d'affichage et la valeur de t_{peak} reflètent la position du curseur.

```
public JSlider slider() {
    if (slider == null) {
        slider = new JSlider();
        sliderMax = slider.getMaximum();
        sliderMin = slider.getMinimum();
        slider.addChangeListener( ?? );
        slider.setValue(slider.getMinimum());
    }
    return slider;
}

public void stateChanged(ChangeEvent e) {
    double val = slider.getValue();
    double tp = (val - sliderMin) / (sliderMax - sliderMin);
    burnPanel(). ?? ( ?? );
    thrustPanel(). ?? ( ?? );
    valueLabel(). ?? ( ?? );
}
```

- Les solutions des exercices de ce chapitre sont données dans l'Annexe B.

La classe `ShowBallistics` actualise les objets pour les deux panneaux d'affichage et la valeur de t_{peak} , qui dépend de la valeur du curseur. C'est une pratique fréquente et pas nécessairement mauvaise, mais notez qu'elle défait totalement l'objectif de `OBSERVER`. Swing applique `OBSERVER` pour que le curseur n'ait pas à savoir quels sont les clients intéressés par son état. L'application `ShowBallistics` nous place toutefois dans la situation que voulions éviter, à savoir : un seul objet, l'application, sait quels objets actualiser et se charge d'émettre les interrogations appropriées au lieu de laisser chaque objet s'enregistrer lui-même de manière individuelle.

Pour créer un `OBSERVER` d'une plus grande granularité, vous pouvez apporter quelques changements au code pour laisser chaque composant intéressé s'enregistrer lui-même pour recevoir les événements de changement du curseur.

Dans cette conception, vous pouvez déplacer les appels de `addChangeListener()` se trouvant dans la méthode `slider()` vers les constructeurs des composants dépendants :

```
public BallisticsPanel2(  
    BallisticsFunction func,  
    JSlider slider) {  
    this.func = func;  
    this.slider = slider;  
    slider.addChangeListener(this);  
}
```

Exercice 9.2

Produisez un nouveau diagramme de classes pour une conception laissant chaque objet intéressé s'enregistrer pour recevoir les événements du curseur. Veillez à tenir compte du champ affichant la valeur du curseur.

Lorsque le curseur change, l'objet `BallisticsPanel2` en est averti. Le champ `tPeak` recalcule sa valeur et se redessine :

```
public void stateChanged(ChangeEvent e) {  
    double val = slider.getValue();  
    double max = slider.getMaximum();  
    double min = slider.getMinimum();  
    tPeak = (val - min) / (max - min);  
    repaint();  
}
```

Cette nouvelle conception donne lieu à un nouveau problème. Chaque objet intéressé s'enregistre et s'actualise lors des changements du curseur. Cette répartition de la responsabilité est bonne, mais chaque composant qui est à l'écoute des événements du curseur doit recalculer la valeur de `tPeak`. En particulier, si vous utilisez une classe `BallisticsLabel2` — comme dans la solution de l'Exercice 9.2 —, sa méthode `stateChanged()` sera presque identique à la méthode `stateChanged()` de `BallisticsPanel2`. Pour réduire ce code dupliqué, nous pouvons extraire un objet de domaine sous-jacent à partir de la présente conception.

Nous pouvons simplifier le système en introduisant une classe `Tpeak` qui contiendra la valeur de temps critique. L'application restera alors à l'écoute des événements du curseur et actualisera un objet `Tpeak` auquel seront attentifs les autres composants intéressés. Cette approche devient une conception **MVC** (*Modèle-Vue-Contrôleur*) (voir [Buschmann et al. 1996] pour un traitement en détail de l'approche MVC).

Modèle-Vue-Contrôleur

A mesure que les applications et les systèmes augmentent de taille, il est important de répartir toujours davantage les responsabilités pour que les classes et les packages restent d'une taille suffisamment petite pour faciliter la maintenance. La triade Modèle-Vue-Contrôleur sépare un objet intéressant, le modèle, des éléments de GUI qui le représentent et le manipulent, la vue et le contrôleur. Java gère cette séparation au moyen de listeners, mais comme le montre la section précédente, toutes les conceptions recourant à des listeners ne suivent pas nécessairement une approche MVC.

Les versions initiales de l'application `ShowBallistics` combinent la logique intelligente d'une interface GUI d'application et des informations balistiques. Vous pouvez réduire ce code par une approche MVC pour redistribuer les responsabilités de l'application. Dans le processus de recodage, la classe `ShowBallistics` révisée garde les vues et les contrôleurs dans ses éléments de GUI.

L'idée des créateurs de MVC était que l'apparence d'un composant (la vue) pouvait être séparée de ce qui l'animait (le contrôleur). Dans la pratique, l'apparence d'un composant de GUI et ses fonctionnalités supportant l'interaction utilisateur sont étroitement couplées, et l'emploi typique de Swing ne sépare pas les vues des contrôleurs — si vous vous plongez davantage dans les rouages internes de ces concepts dans Swing, vous verrez émerger cette séparation. La valeur de MVC réside dans le fait d'extraire le *modèle* d'une application pour le placer dans un domaine propre.

Le *modèle* dans l'application `ShowBallistics` est la valeur `tPeak`. Pour recoder avec l'approche MVC, nous pourrions introduire une classe `Tpeak` qui contiendrait cette valeur de temps crête et autoriser les listeners intéressés à s'enregistrer pour recevoir les événements de changement. Une telle classe pourrait ressembler à l'extrait suivant :

```
package app.observer.ballistics3;
import java.util.Observable;

public class Tpeak extends Observable {
    protected double value;
    public Tpeak(double value) {
        this.value = value;
    }

    public double getValue() {
        return value;
    }
}
```

```
    }

    public void setValue(double value) {
        this.value = value;
        setChanged();
        notifyObservers();
    }
}
```

Si vous deviez réviser ce code chez Oozinoz, un point essentiel serait soulevé : presque aucune portion de ce code ne se rapporte au moment où le taux de combustion atteint sa valeur crête. En fait, cet extrait ressemble à un outil relativement générique pour contenir une valeur et pour alerter des listeners lorsqu'elle change. Nous pourrions modifier le code pour éliminer ce caractère générique, mais examinons tout d'abord une conception révisée utilisant la classe Tpeak.

Nous pouvons maintenant élaborer une conception dans laquelle l'application reste attentive au curseur et tous les autres composants restent à l'écoute de l'objet Tpeak. Lorsque le curseur est déplacé, l'application change la valeur dans l'objet Tpeak. Les panneaux d'affichage et le champ de valeur sont à l'écoute de cet objet et s'actualisent lorsqu'il change. Les classes BurnRate et Thrust emploient l'objet Tpeak pour le calcul de leurs fonctions, mais elles n'ont pas besoin d'écouter les événements (c'est-à-dire de s'enregistrer à cet effet).

Exercice 9.3

Créez un diagramme de classes montrant l'application dépendant du curseur alors que les panneaux d'affichage et le champ de valeur dépendent d'un objet Tpeak.

Cette conception permet de n'effectuer qu'une seule fois le travail de traduction de la valeur du curseur en valeur de temps crête. L'application actualise un seul objet Tpeak, et tous les objets de GUI intéressés par un changement peuvent interroger l'objet pour en connaître la nouvelle valeur.

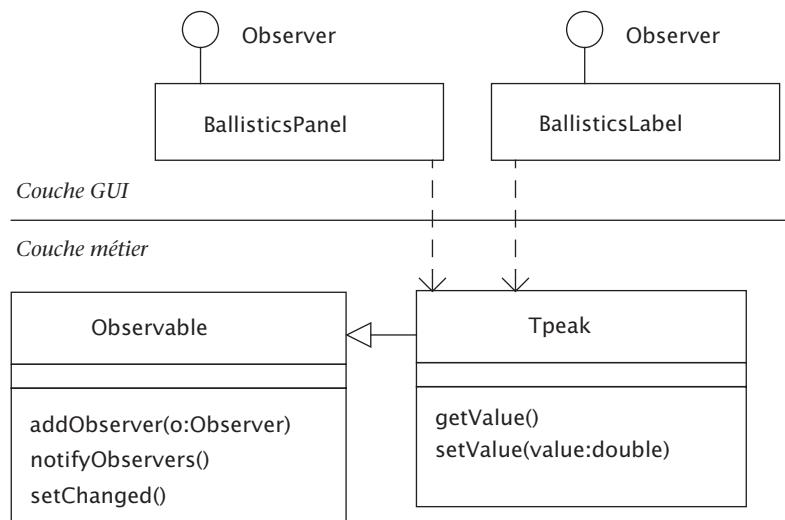
La classe Tpeak ne fait pas que conserver une valeur. Aussi essayons-nous de recoder l'application pour créer une classe conteneur de valeur. De plus, il est possible qu'un nombre observé, tel qu'une valeur de temps crête, ne soit pas une valeur isolée mais plutôt l'attribut d'un objet de domaine. Par exemple, le temps crête est un attribut d'un moteur de fusée. Nous pouvons tenter d'améliorer notre conception

pour séparer les classes, avec une classe permettant aux objets de GUI d'observer les objets de domaine.

Lorsque vous décidez de séparer des objets de GUI d'objets de domaine, ou **d'objets métiers** (*business object*), vous pouvez créer des couches de code. Une *couche* est un groupe de classes ayant des responsabilités similaires, souvent rassemblées dans un seul package Java. Les couches supérieures, telles qu'une couche GUI, dépendent généralement seulement de classes situées dans des couches de niveau égal ou inférieur. Le codage en couches demande généralement d'avoir une définition claire des interfaces entre les couches, telles qu'entre une GUI et les objets métiers qu'elle représente. Vous pouvez réorganiser les responsabilités du code de `ShowBallistics` pour obtenir un système en couches, comme le montre la Figure 9.3.

Figure 9.3

En créant une classe `Tpeak observable`, vous pouvez séparer la logique métier et la couche GUI.



La conception illustrée Figure 9.3 crée une classe `Tpeak` pour modéliser la valeur t_{peak} critique pour les résultats des équations balistiques affichés par l'application. Les classes `BallisticsPanel` et `BallisticsLabel` dépendent de `Tpeak`. Plutôt que de laisser à l'objet `Tpeak` la responsabilité d'actualiser les éléments de GUI, la conception applique le pattern OBSERVER pour que les objets intéressés puissent s'enregistrer pour être notifiés de tout changement de `Tpeak`. Les bibliothèques de classes Java offrent un support en fournissant une classe `Observable` et une

interface `Observer` contenues dans le package `java.util` package. La classe `Tpeak` peut dériver une sous-classe `Observable` et actualiser ses objets "observateurs" lorsque sa valeur change :

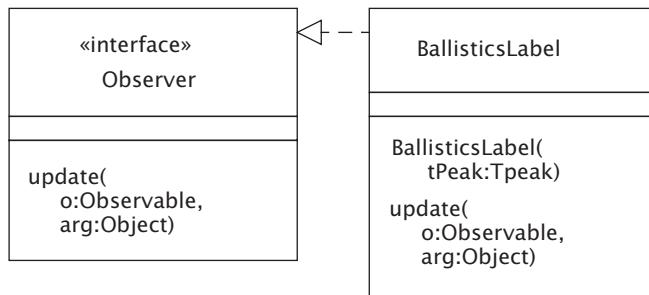
```
public void setValue(double value) {
    this.value = value;
    setChanged();
    notifyObservers();
}
```

Notez que vous devez appeler `setChanged()` de façon que la méthode `notifyObservers()`, héritée d`Observable`, signale le changement.

La méthode `notifyObservers()` invoque la méthode `update()` de chaque observateur enregistré. La méthode `update()` est une exigence pour les classes implémentant l'interface `Observer`, comme illustré Figure 9.4.

Figure 9.4

Un objet `BallisticsLabel` est un `Observer`. Il peut s'enregistrer auprès d'un objet `Observable` comme objet intéressé pour que la méthode `update()` de l'objet label soit appelée lorsque l'objet `Observable` change.



Un objet `BallisticsLabel` n'a pas besoin de conserver une référence vers l'objet `Tpeak` qu'il observe. Au lieu de cela, le constructeur de `BallisticsLabel` peut s'enregistrer pour obtenir les mises à jour lorsque l'objet `Tpeak` change. La méthode `update()` de l'objet `BallisticsLabel` recevra l'objet `Tpeak` en tant qu'argument `Observable`. La méthode peut transtyper (*cast*) l'argument en `Tpeak`, extraire la nouvelle valeur, modifier le champ de valeur et redessiner l'écran.

Exercice 9.4

Rédigez le code complet pour `BallisticsLabel.java`.

La nouvelle conception de l'application de calcul balistique sépare l'objet métier des éléments de GUI qui le représentent. Deux exigences doivent être respectées pour qu'elle fonctionne.

1. Les implémentations de `Observer` doivent s'enregistrer pour signaler leur intérêt et doivent s'actualiser elles-mêmes de façon correcte, souvent en incluant l'actualisation de l'affichage.
2. Les sous-classes de `Observable` doivent notifier les objets intéressés observateurs lorsque leurs valeurs changent.

Ces deux étapes définissent la plupart des interactions dont vous avez besoin entre les différentes couches de l'application balistique. Il vous faut aussi prévoir un objet `Tpeak` qui change conformément lorsque la position du curseur change. Vous pouvez pour cela instancier une sous-classe anonyme de `ChangeListener`.

Exercice 9.5

Supposez que `tPeak` soit une instance de `Tpeak` et un attribut de la classe `ShowBallistics3`. Complétez le code de `ShowBallistics3.slider()` de façon que le changement du curseur actualise `tPeak`.

```
public JSlider slider() {
    if (slider == null) {
        slider = new JSlider();
        sliderMax = slider.getMaximum();
        sliderMin = slider.getMinimum();
        slider.addChangeListener
        (
            new ChangeListener()
            {
                // Exercice !
            }
        );
        slider.setValue(slider.getMinimum());
    }
    return slider;
}
```

Lorsque vous appliquez l'approche MVC, le flux des événements peut sembler indirect. Un mouvement de curseur dans l'application de calculs balistiques provoque l'actualisation d'un objet `Tpeak` par un objet `ChangeListener`. En retour, un changement dans l'objet `Tpeak` informe les objets d'affichage (champ et panneaux) qui actualisent leur affichage. Le changement est répercuté de la couche GUI à la couche métier puis à nouveau vers la couche GUI.

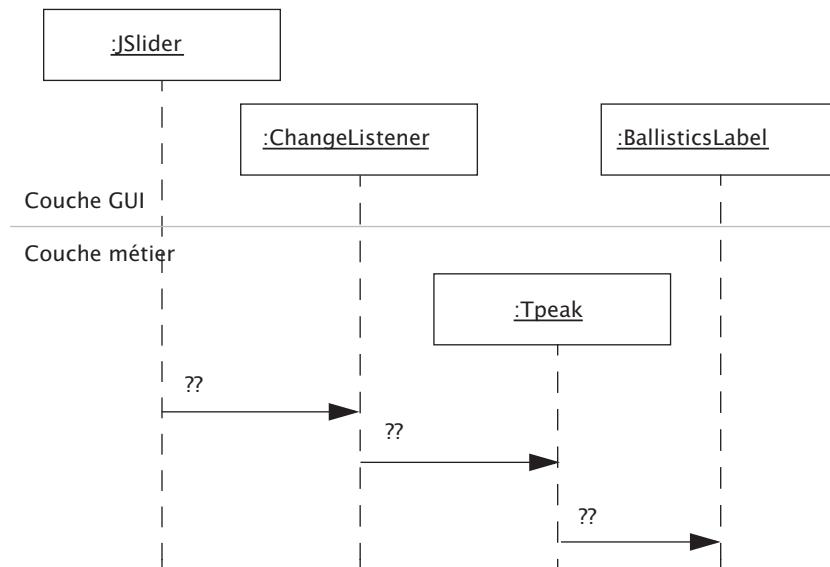


Figure 9.5

L'approche MVC demande de passer les messages de la couche GUI vers la couche métier puis à nouveau vers la couche GUI.

Le bénéfice d'une telle conception en couches réside dans la valeur de l'interface et dans le niveau d'indépendance que vous obtenez entre les couches. Ce codage en couches est une distribution en couches des responsabilités, ce qui produit un code plus simple à maintenir. Par exemple, dans notre exemple d'application de calculs balistiques, vous pouvez ajouter une seconde GUI, peut-être pour un équipement portable, sans avoir à changer les classes de la couche objets métiers. De même, vous pourriez ajouter dans cette dernière une nouvelle source de changement qui actualise un objet `Tpeak`. Dans ce cas, le mécanisme `OBSERVER` déjà en place met automatiquement à jour les objets de la couche GUI.

Cette conception en couches rend aussi possible l'exécution de différentes couches sur différents ordinateurs. Une couche ou un ensemble de couches s'exécutant sur un ordinateur constitue un niveau dans un système multiniveau (*n-tier*). Une conception multiniveau peut réduire la quantité de code devant être exécutée sur l'ordinateur de l'utilisateur final. Elle vous permet aussi d'apporter des changements dans les classes métiers sans avoir à changer le logiciel sur les machines des utilisateurs, ce qui simplifie grandement le déploiement. Toutefois, l'échange de messages entre ordinateurs a son coût et le déploiement en environnement multiniveau doit être fait judicieusement. Par exemple, vous ne pourrez probablement pas vous permettre de faire attendre l'utilisateur pendant que les événements de défilement transitent entre son ordinateur et le serveur. Dans ce cas, vous devrez probablement laisser le défilement se produire sur la machine de l'utilisateur, puis concevoir sous forme d'une autre action utilisateur distincte la validation d'une nouvelle valeur de temps crête. En bref, **OBSERVER** supporte l'architecture MVC, ce qui promeut la conception en couches et s'accompagne de nombreux avantages pour le développement et le déploiement de logiciels.

Maintenance d'un objet **Observable**

Vous ne pourrez pas toujours créer la classe que vous voulez pour guetter les changements d'une sous-classe de **Observable**. En particulier, votre classe peut déjà être une sous-classe de quelque chose d'autre que **Object**. Dans ce cas, vous pouvez associer à votre classe un objet **Observable** et faire en sorte qu'elle lui transmette les appels de méthodes essentiels. La classe **Component** dans `java.awt` suit cette approche mais utilise un objet **PropertyChangeSupport** à la place d'un objet **Observable**.

La classe **PropertyChangeSupport** est semblable à la classe **Observable**, mais elle fait partie du package `java.beans`. L'API JavaBeans permet la création de composants réutilisables. Elle trouve sa plus grande utilité dans le développement de composants de GUI, mais vous pouvez certainement l'appliquer à d'autres fins. La classe **Component** emploie un objet **PropertyChangeSupport** pour permettre aux objets observateurs intéressés de s'enregistrer et de recevoir une notification de changement des propriétés de champs, de panneaux, et d'autres éléments de GUI. La Figure 9.6 montre la relation qui existe entre la classe **Component** de `java.awt` et la classe **PropertyChangeSupport**.

La classe `PropertyChangeSupport` illustre un problème qu'il vous faudra résoudre lors de l'emploi du pattern OBSERVER, à savoir, le niveau de détails à fournir par la classe observée pour indiquer ce qui a changé. Cette classe utilise une approche *push*, où le modèle renseigne sur ce qui s'est produit — dans `PropertyChangeSupport`, la notification indique le changement de la propriété, d'une ancienne valeur à une nouvelle valeur. Une autre option est l'approche *pull*, où le modèle signale aux objets observateurs qu'il a changé, mais ceux-ci doivent interroger le modèle pour savoir de quelle manière. Les deux approches peuvent être appropriées. L'approche *push* peut signifier davantage de travail de développement et associe étroitement les objets observateurs à l'objet observé, mais offre la possibilité de meilleures performances.

La classe `Component` duplique une partie de l'interface de la classe `PropertyChangeSupport`. Ces méthodes dans `Component` transmettent chacune l'appel de message vers une instance de la classe `PropertyChangeSupport`.

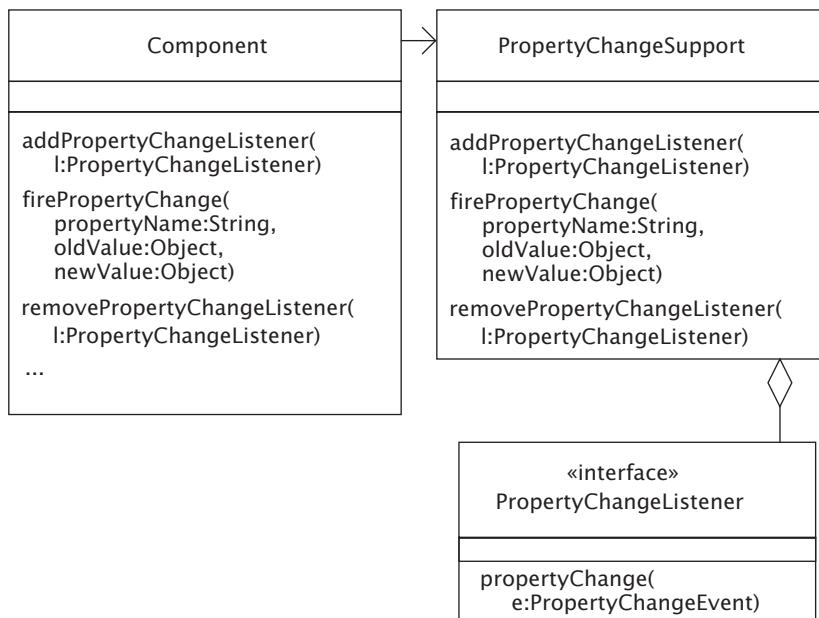


Figure 9.6

Un objet `Component` renseigne un objet `PropertyChangeSupport` qui renseigne un ensemble de listeners.

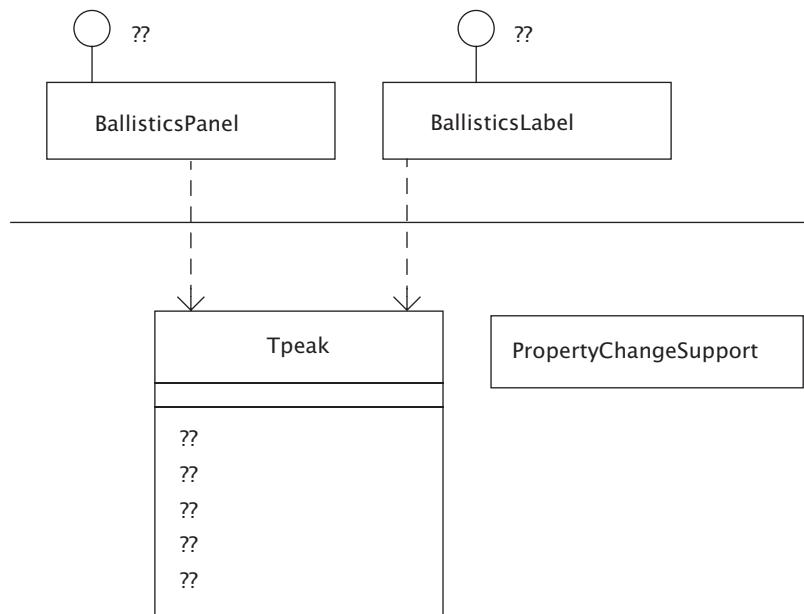


Figure 9.7

Un objet métier **Tpeak** peut déléguer les appels qui affectent les listeners à un objet **PropertyChangeSupport**.

Exercice 9.7

Complétez le diagramme de classes de la Figure 9.7 pour que **Tpeak** utilise un objet **PropertyChangeSupport** afin de gérer des listeners.

Que vous utilisiez **Observer**, **PropertyChangeSupport** ou une autre classe pour appliquer le pattern **OBSERVER**, l'important est de définir une dépendance un-à-plusieurs entre des objets. Lorsque l'état d'un objet change, tous les objets dépendants en sont avertis et sont actualisés automatiquement. Cela limite la responsabilité et facilite la maintenance des objets intéressants et de leurs observateurs intéressés.

Résumé

Le pattern OBSERVER apparaît fréquemment dans les applications avec GUI et constitue un pattern fondamental dans les bibliothèques de GUI Java. Avec ces composants, vous n'avez jamais besoin de modifier ou de dériver une sous-classe d'une classe de composant simplement pour communiquer ses événements à d'autres objets intéressés. Pour de petites applications, une pratique courante consiste à n'enregistrer qu'un seul objet, l'application, pour recevoir les événements d'une GUI. Il n'y a pas de problème inhérent à cette approche, mais sachez qu'elle inverse la répartition des responsabilités que vise OBSERVER. Pour une grande GUI, envisagez la possibilité de passer à une conception MVC, en permettant à chaque objet intéressé de gérer son besoin d'être notifié au lieu d'introduire un objet central médiateur. L'approche MVC vous permet aussi d'associer avec davantage de souplesse diverses couches de l'application, lesquelles peuvent alors changer de façon indépendante et être exécutées sur des machines différentes.

10

MEDIATOR

Le développement orienté objet ordinaire distribue la responsabilité aussi loin que possible, avec chaque objet accomplissant sa tâche indépendamment des autres. Par exemple, le pattern **OBSERVER** supporte cette distribution en limitant la responsabilité d'un objet que d'autres objets trouvent intéressant. Le pattern **SINGLETON** résiste à la distribution de la responsabilité et vous permet de la centraliser au niveau de certains objets que les clients localisent et réutilisent. A l'instar de **SINGLETON**, le pattern **MEDIATOR** centralise la responsabilité mais pour un ensemble spécifique d'objets plutôt que pour tous les clients dans un système.

Lorsque les interactions entre les objets reposent sur une condition complexe impliquant que chaque objet d'un groupe connaisse tous les autres, il est utile d'établir une autorité centrale. La centralisation de la responsabilité est également utile lorsque la logique entourant les interactions des objets en relation est indépendante de l'autre comportement des objets.

L'objectif du pattern MEDIATOR est de définir un objet qui encapsule la façon dont un ensemble d'objets interagissent. Cela promeut un couplage lâche, évitant aux objets d'avoir à se référer explicitement les uns aux autres, et permet de varier leur interaction indépendamment.

Un exemple classique : médiateur de GUI

C'est probablement lors du développement d'une application GUI que vous rencontrerez le plus le pattern **MEDIATOR**. Le code d'une telle application tend à devenir volumineux, demandant à être refactorisé en d'autres classes. La classe `ShowFlight` dans le Chapitre 4, consacré au pattern **FAÇADE**, remplissait initialement

trois rôles. Avant qu'elle ne soit refactorisée, elle servait de panneau d'affichage, d'application GUI complète et de calculateur de trajectoire de vol. La refactorisation a permis de simplifier l'application invoquant l'affichage du panneau de trajectoire, la ramenant à seulement quelques lignes de code. Toutefois, les grosses applications peuvent conserver leur complexité après ce type de refactorisation, même si elles n'incluent que la logique qui crée les composants et définit leur interaction. Considérez l'application de la Figure 10.1.

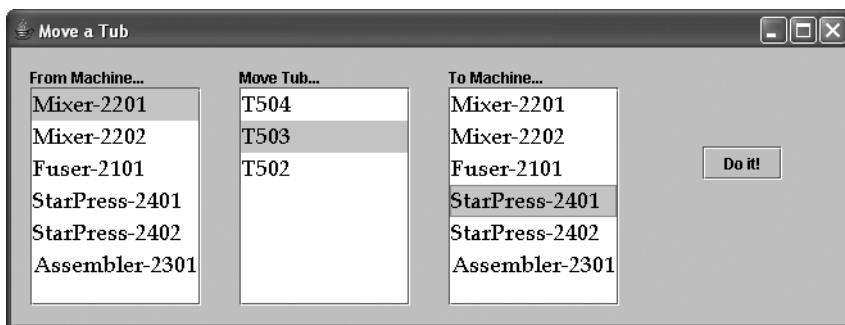


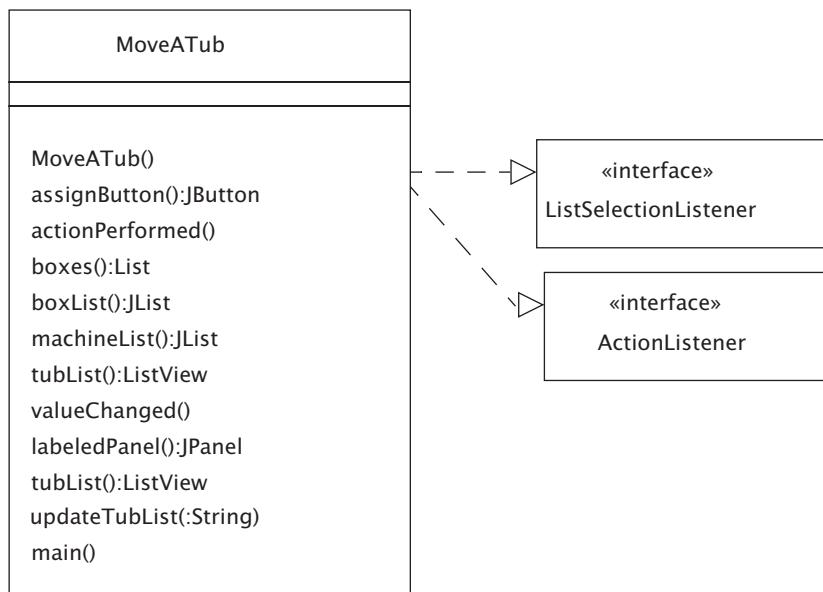
Figure 10.1

Cette application laisse l'utilisateur actualiser manuellement l'emplacement d'un bac de produit chimique.

Oozinoz stocke les produits chimiques dans des bacs (*tub*) en plastique. Les machines lisent les codes-barres sur les bacs pour garder trace de leur emplacement dans l'usine. Parfois, une correction manuelle est nécessaire, notamment lorsqu'un employé déplace un bac au lieu d'attendre qu'un robot le transfère. La Figure 10.1 présente une nouvelle application partiellement développée qui permet à l'utilisateur de spécifier la machine au niveau de laquelle un bac se situe.

Dans l'application `MoveATub`, du package `app.mediator.moveATub`, lorsque l'utilisateur sélectionne une des machines dans la liste de gauche, la liste de bacs change pour afficher ceux présents sur la machine en question. Il peut ensuite sélectionner un des bacs, choisir une machine cible, et cliquer sur le bouton `Do it!` pour actualiser l'emplacement du bac. La Figure 10.2 présente un extrait de la classe de l'application.

Le développeur de cette application l'a créée initialement à l'aide d'un assistant et a commencé à la refactoriser. Environ la moitié des méthodes de `MoveATub` existent

**Figure 10.2**

La classe MoveATub combine des méthodes de création de composants, de gestion d'événements, et de base de données fictive.

pour procéder à une initialisation paresseuse des variables contenant les composants GUI de l'application. La méthode `assignButton()` est un exemple typique :

```

private JButton assignButton() {
    if (assignButton == null) {
        assignButton = new JButton("Do it!");
        assignButton.setEnabled(false);
        assignButton.addActionListener(this);
    }
    return assignButton;
}
  
```

Le programmeur a déjà éliminé les valeurs codées en dur générées par l'assistant pour spécifier l'emplacement et la taille du bouton. Mais un problème plus immédiat est que la classe `MoveATub` comporte un grand nombre de méthodes ayant des utilités différentes.

La plupart des méthodes statiques fournissent une base de données fictive de noms de bacs et de noms de machines. Le développeur envisage de remplacer l'approche consistant à utiliser uniquement des noms par l'emploi d'objets `Tub` et `Machine`.

La majorité des autres méthodes contient la logique de gestion des événements de l'application. Par exemple, la méthode `valueChanged()` détermine si le bouton d'assignation a été activé :

```
public void valueChanged(ListSelectionEvent e) {  
    // ...  
    assignButton().setEnabled(  
        ! tubList().isSelectionEmpty()  
        && ! machineList().isSelectionEmpty());  
}
```

Le développeur pourrait placer la méthode `valueChanged()` et les autres méthodes de gestion d'événements dans une classe médiateur distincte. Il convient de noter que le pattern MEDIATOR est déjà à l'œuvre dans la classe `MoveATub` : les composants ne s'actualisent pas directement les uns les autres. Par exemple, ni la machine ni les composants liste n'actualisent directement le bouton d'assignation. A la place, l'application `MoveATub` enregistre des listeners pour les événements de sélection puis actualise le bouton, selon les éléments sélectionnés dans les deux listes. Dans cette application, un objet `MoveATub` agit en tant que médiateur, recevant les événements et dispatchant les actions correspondantes.

Les bibliothèques de classes Java, ou JCL (*Java Class Libraries*), vous incitent à utiliser un médiateur mais n'imposent aucunement que l'application soit son propre médiateur. Au lieu de mélanger dans une même classe des méthodes de création de composants, des méthodes de gestion d'événements et des méthodes de base de données fictive, il serait préférable de les placer dans des classes avec des spécialisations distinctes.

La refactorisation donne au médiateur une classe propre, vous permettant de le développer et de vous concentrer dessus séparément. Lorsque l'application refactorisée s'exécute, les composants passent les événements à un objet `MoveATub-Mediator`. Le médiateur peut avoir une action sur des objets non-GUI, par exemple pour actualiser la base de données lorsqu'une assignation a lieu. Il peut aussi rappeler des composants GUI, par exemple pour désactiver le bouton à l'issue de l'assignation.

Les composants GUI pourraient appliquer le pattern MEDIATOR automatiquement, signalant à un médiateur lorsque des événements surviennent plutôt que de prendre la responsabilité d'actualiser directement d'autres composants. Les applications GUI donnent probablement lieu à l'exemple le plus courant de MEDIATOR, mais il existe d'autres situations où vous pourriez vouloir introduire un médiateur.

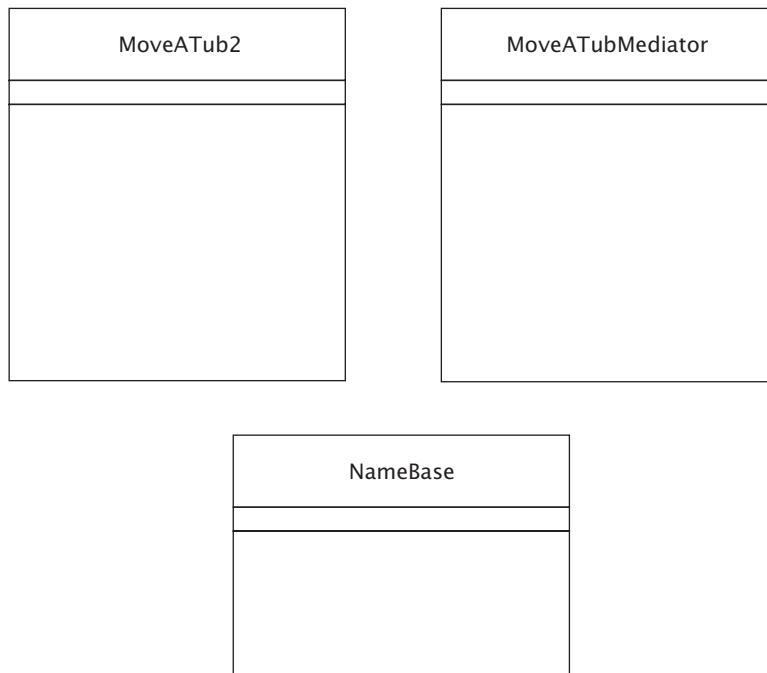


Figure 10.3

Séparation des méthodes de création de composants, des méthodes de gestion d'événements et des méthodes de base de données fictive de l'application.

Exercice 10.2

Dessinez un diagramme illustrant ce qui se produit lorsque l'utilisateur clique sur le bouton Do it!. Montrez quels objets sont d'après vous les plus importants ainsi que les messages échangés entre ces objets.

Lorsque l'interaction d'un ensemble d'objets est complexe, vous pouvez centraliser la responsabilité de cette interaction dans un objet médiateur qui reste extérieur au groupe. Cela promeut le **couplage lâche** (*loose coupling*), c'est-à-dire une réduction de la responsabilité que chaque objet entretient vis-à-vis de chaque autre. Gérer cette interaction dans une classe indépendante présente aussi l'avantage de simplifier et de standardiser les règles d'interaction. La valeur d'un médiateur apparaît de manière évidente lorsque vous avez besoin de gérer l'intégrité relationnelle.

Médiateur d'intégrité relationnelle

Le paradigme orienté objet doit sa puissance en partie au fait qu'il permet de représenter aisément au moyen d'objets Java les relations entre des objets du monde réel. Toutefois, la capacité d'un modèle objet Java à refléter le monde réel se heurte à deux limitations. Premièrement, les objets réels varient avec le temps et Java n'offre aucun support intégré pour cela. Par exemple, les instructions d'assignation éliminent toute valeur précédente au lieu de la mémoriser, comme un être humain le ferait. Deuxièmement, dans le monde réel, les relations sont aussi importantes que les objets, alors qu'elles ne bénéficient que d'un faible support dans les langages orientés objet actuels, Java y compris. Par exemple, il n'y a pas de support intégré pour le fait que si la machine Star Press 2402 se trouve dans la travée 1, la travée 1 doit contenir la machine Star Press 2402. En fait, de telles relations risquent d'être ignorées, d'où l'intérêt d'appliquer le pattern MEDIATOR.

Considérez les bacs (*tub*) en plastique d'Oozinoz. Ces bacs sont toujours assignés à une certaine machine. Vous pouvez modéliser cette relation au moyen d'une table, comme l'illustre le Tableau 10.1.

Tableau 10.1 : Enregistrer les informations relationnelles dans une table préserve l'intégrité relationnelle

Bac	Machine
T305	StarPress-2402
T308	StarPress-2402
T377	ShellAssembler-2301
T379	ShellAssembler-2301
T389	ShellAssembler-2301
T001	Fuser-2101
T002	Fuser-2101

Le Tableau 10.1 illustre la **relation** entre les bacs et les machines, c'est-à-dire leur positionnement réciproque. Mathématiquement, une relation est un sous-ensemble de toutes les paires ordonnées d'objets, tel qu'il y a une relation des bacs vers les machines et une relation des machines vers les bacs. L'unicité des valeurs de la

colonne Bac garantit qu'aucun bac ne peut apparaître sur deux machines à la fois. Voyez l'encadré suivant pour une définition plus stricte de la cohérence relationnelle dans un modèle objet.

Intégrité relationnelle

Un modèle objet présente une *cohérence relationnelle* si chaque fois que l'objet a pointe vers l'objet b, l'objet b pointe vers l'objet a.

Pour une définition plus rigoureuse, considérez deux classes, Alpha et Beta. A représente l'ensemble des objets qui sont des instances de la classe Alpha, et B représente l'ensemble des objets qui sont des instances de la classe Beta. a et b sont donc des membres respectivement de A et de B, et la *paire ordonnée* (a, b) indique que l'objet $a \in A$ possède une référence vers l'objet $b \in B$. Cette référence peut soit être directe soit faire partie d'un ensemble de références, comme lorsque l'objet a possède un objet List qui inclut b.

Le *produit cartésien* $A \times B$ est l'ensemble de toutes les paires ordonnées possibles (a, b) avec $a \in A$ et $b \in B$. Les ensembles A et B autorisent les deux produits cartésiens $A \times B$ et $B \times A$. Une *relation de modèle objet* sur A et B est le sous-ensemble de $A \times B$ qui existe dans un modèle objet. AB représente ce sous-ensemble, et BA représente le sous-ensemble $B \times A$ qui existe dans le modèle.

Toute relation binaire $R \subseteq A \times B$ possède un *inverse* $R^{-1} \subseteq B \times A$ défini par :

$$(b, a) \in R^{-1} \text{ si et seulement si } (a, b) \in R$$

L'inverse de AB fournit l'ensemble des références qui doivent exister de B vers les instances de A lorsque le modèle objet est cohérent. Autrement dit, les instances des classes Alpha et Beta sont cohérentes relationnellement si et seulement si BA est l'inverse de AB.

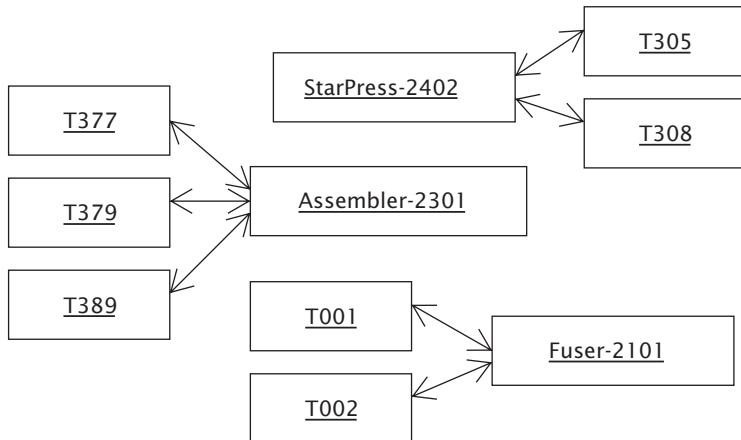
Lorsque vous enregistrez les informations relationnelles des bacs et des machines dans une table, vous pouvez garantir que chaque bac se trouve sur une seule machine à la fois en appliquant comme restriction qu'il n'apparaisse qu'une seule fois dans la colonne Bac. Une façon de procéder est de définir cette colonne comme clé primaire de la table dans une base de données relationnelle. Avec ce modèle, qui reflète la réalité, un bac ne peut pas apparaître sur deux machines en même temps : $(b, a) \in R^{-1}$ si et seulement si $(a, b) \in R$.

Un modèle objet ne peut pas garantir l'intégrité relationnelle aussi facilement qu'un modèle relationnel. Considérez l'application MoveATub. Comme évoqué précédemment, son concepteur prévoit d'abandonner l'emploi de noms au profit d'objets Tub et Machine. Lorsqu'un bac se trouve près d'une machine, l'objet qui le représente possède une référence vers l'objet représentant la machine. Chaque objet Machine

possède une collection d'objets `Tub` représentant les bacs situés près de la machine. La Figure 10.4 illustre un modèle objet typique.

Figure 10.4

Un modèle objet distribue les informations sur les relations.

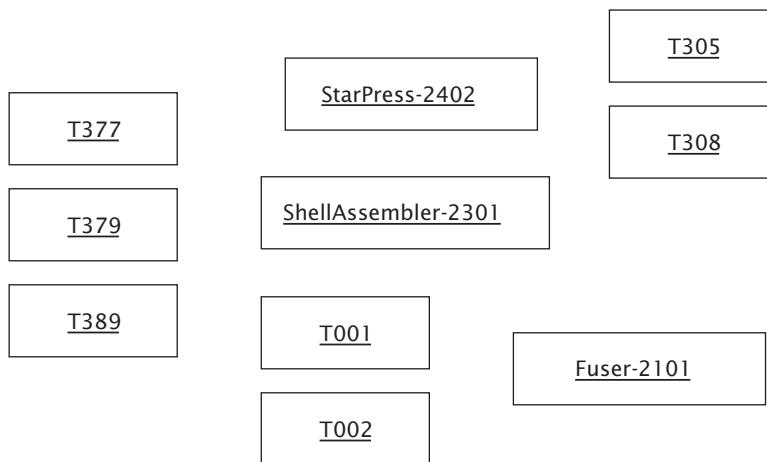


Les doubles flèches de cette figure mettent en évidence le fait que les bacs ont connaissance des machines et vice versa. Les informations sur cette relation bac/machine sont maintenant distribuées à travers de nombreux objets au lieu de figurer dans une table centrale, ce qui la rend plus difficile à gérer et fait qu'elle se prête bien à l'application du pattern MEDIATOR.

Considérez une anomalie survenue chez Oozinoz lorsqu'un développeur a commencé à modéliser une nouvelle machine incluant un lecteur de codes-barres pour identifier les bacs. Après scannage d'un bac `t` pour obtenir son identifiant, son emplacement est défini comme étant sur la machine `m` au moyen du code suivant :

```
// Renseigne le bac sur la machine et inversement
t.setMachine(m);
m.addTub(t);
```

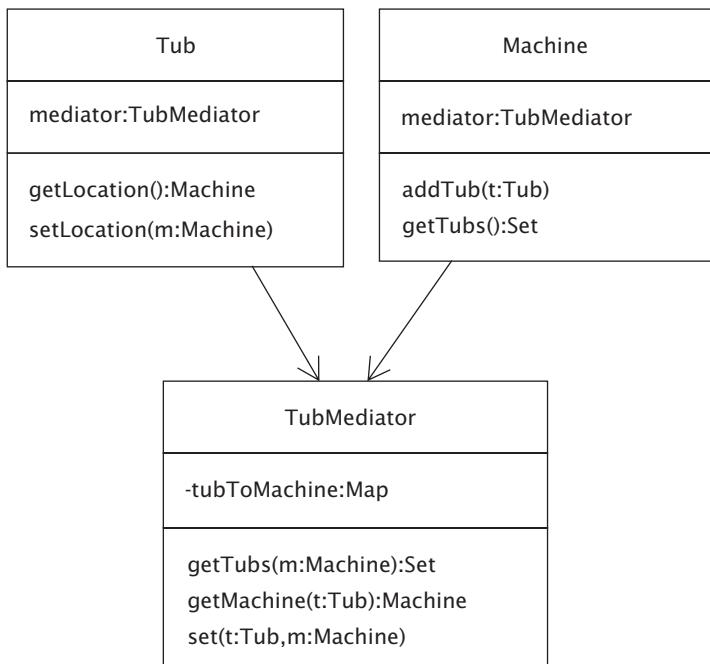
Le moyen le plus simple de garantir l'intégrité relationnelle est de replacer les informations relationnelles dans une seule table gérée par un objet médiateur. Au lieu que les machines aient connaissance des bacs et inversement, il faut donner à ces objets une référence vers un médiateur qui s'occupe de gérer la table. Cette table peut être une instance de la classe `Map` (du package `java.util`). La Figure 10.6 présente un diagramme de classe incluant un médiateur.

**Figure 10.5**

Une fois complété, ce diagramme mettra en évidence l'erreur dans le code qui actualise l'emplacement du bac.

Figure 10.6

Les objets Tub et Machine s'appuient sur un médiateur pour contrôler la relation entre les bacs et les machines.



La classe `Tub` possède un attribut d'emplacement qui permet d'enregistrer la machine à proximité de laquelle se trouve un bac. Le code garantit qu'un bac ne peut être qu'à un seul endroit à la fois, utilisant un objet `TubMediator` pour gérer la relation bac/machine :

```
package com.oozinoz.machine;
public class Tub {
    private String id;
    private TubMediator mediator = null;

    public Tub(String id, TubMediator mediator) {
        this.id = id;
        this.mediator = mediator;
    }

    public Machine getLocation() {
        return mediator.getMachine(this);
    }

    public void setLocation(Machine value) {
        mediator.set(this, value);
    }

    public String toString() {
        return id;
    }

    public int hashCode() {
        // ...
    }

    public boolean equals(Object obj) {
        // ...
    }
}
```

La méthode `setLocation()` de la classe `Tub` utilise un médiateur pour actualiser l'emplacement d'un bac, lui déléguant la responsabilité de préserver l'intégrité relationnelle. Cette classe implémente les méthodes `hashCode()` et `equals()` de sorte que les objets `Tub` puissent être correctement stockés dans une table de hachage. Voici les détails du code :

```
public int hashCode() {
    return id.hashCode();
}

public boolean equals(Object obj) {
```

```
if (obj == this) return true;  
  
if (obj.getClass() != Tub.class)  
    return false;  
  
Tub that = (Tub) obj;  
return id.equals(that.id);  
}
```

La classe `TubMediator` utilise un objet `Map` pour stocker la relation bac/machine. Le médiateur peut ainsi garantir que le modèle objet n'autorise jamais deux machines à posséder le même bac :

```
public class TubMediator {  
    protected Map tubToMachine = new HashMap();  
  
    public Machine getMachine(Tub t) {  
        // Exercice !  
    }  
  
    public Set getTubs(Machine m) {  
        Set set = new HashSet();  
        Iterator i = tubToMachine.entrySet().iterator();  
        while (i.hasNext()) {  
            Map.Entry e = (Map.Entry) i.next();  
            if (e.getValue().equals(m))  
                set.add(e.getKey());  
        }  
        return set;  
    }  
  
    public void set(Tub t, Machine m) {  
        // Exercice !  
    }  
}
```

Exercice 10.4

Ecrivez le code des méthodes `getMachine()` et `set()` de la classe `TubMediator`.

Plutôt que d'introduire une classe médiateur, vous pourriez garantir qu'un bac ne se trouve jamais sur deux machines en même temps en plaçant la logique directement dans les classes `Tub` et `Machine`. Toutefois, cette logique préserve l'intégrité relationnelle et n'a pas grand-chose à voir avec le fonctionnement des bacs et des machines. Elle peut aussi être source d'erreurs. Une erreur possible serait de déplacer

un bac vers une autre machine, en actualisant ces deux objets mais pas la machine précédente.

L'emploi d'un médiateur permet d'encapsuler dans une classe indépendante la logique définissant la façon dont les objets interagissent. Au sein du médiateur, il est facile de s'assurer que le fait de changer l'emplacement d'un objet Tub éloigne automatiquement le bac de la machine sur laquelle il se trouvait. Le code de test JUnit suivant, du package `TubTest.java`, présente ce comportement :

```
public void testLocationChange() {  
    TubMediator mediator = new TubMediator();  
    Tub t = new Tub("T403", mediator);  
    Machine m1 = new Fuser(1001, mediator);  
    Machine m2 = new Fuser(1002, mediator);  
  
    t.setLocation(m1);  
    assertTrue(m1.getTubs().contains(t));  
    assertTrue(!m2.getTubs().contains(t));  
  
    t.setLocation(m2);  
    assertFalse(m1.getTubs().contains(t));  
    assertTrue(m2.getTubs().contains(t));  
}
```

Lorsque vous disposez d'un modèle objet qui n'est pas lié à une base de données relationnelle, vous pouvez utiliser des médiateurs pour préserver l'intégrité relationnelle de votre modèle. Confier la gestion de la relation à des médiateurs permet à ces classes de se spécialiser dans cette préservation.

Exercice 10.5

Pour ce qui est d'extraire une logique d'une classe ou d'une hiérarchie existante afin de la placer dans une nouvelle classe, MEDIATOR ressemble à d'autres patterns. Citez deux autres patterns pouvant impliquer une telle refactorisation.

Résumé

Le pattern MEDIATOR promeut le couplage lâche, évitant à des objets en relation de devoir se référer explicitement les uns aux autres. Il intervient le plus souvent dans le développement d'applications GUI, lorsque vous voulez éviter d'avoir à gérer la complexité liée à l'actualisation mutuelle d'objets. L'architecture de Java vous pousse dans cette direction, vous encourageant à définir des objets qui enregistrent

des listeners pour les événements GUI. Si vous développez des interfaces utilisateur avec Java, vous appliquez probablement ce pattern.

Bien que ce chapitre puisse vous inciter à utiliser le pattern MEDIATOR lors de la création d'une interface GUI, sachez que Java ne vous oblige pas à extraire cette médiation de la classe de l'application. Mais cela peut néanmoins simplifier votre code. Le médiateur peut ainsi se concentrer sur l'interaction entre les composants GUI, et la classe d'application peut se concentrer sur la construction des composants.

D'autres situations se prêtent à l'introduction d'un objet médiateur. Par exemple, vous pourriez en avoir besoin pour centraliser la responsabilité de préserver l'intégrité relationnelle dans un modèle objet. Vous pouvez appliquer MEDIATOR chaque fois que vous devez définir un objet qui encapsule la façon dont un ensemble d'objets interagissent.

PROXY

Un objet ordinaire fait sa part de travail pour supporter l'interface publique qu'il annonce. Il peut néanmoins arriver qu'un objet légitime ne soit pas en mesure d'assumer cette responsabilité ordinaire. Cela peut se produire lorsque l'objet met beaucoup de temps à se charger, lorsqu'il s'exécute sur un autre ordinateur, ou lorsque vous devez intercepter des messages qui lui sont destinés. Dans de telles situations, un objet *proxy* peut prendre cette responsabilité vis-à-vis d'un client et transmettre les requêtes au moment voulu à l'objet cible sous-jacent.

L'objectif du pattern PROXY est de contrôler l'accès à un objet en fournissant un intermédiaire pour cet objet.

Un exemple classique : proxy d'image

Un objet proxy possède généralement une interface qui est quasiment identique à celle de l'objet auquel il sert d'intermédiaire. Il accomplit sa tâche en transmettant lorsqu'il se doit les requêtes à l'objet sous-jacent auquel il contrôle l'accès. Un exemple classique du pattern PROXY intervient pour rendre plus transparent le chargement d'images volumineuses en mémoire. Imaginez que les images d'une application doivent apparaître dans des pages ou panneaux qui ne sont pas affichés initialement. Pour éviter de charger ces images avant qu'elles ne soient requises, vous pourriez leur substituer des proxies qui s'occuperaient de les charger à la demande. Cette section présente un exemple d'un tel proxy. Notez toutefois que les conceptions qui emploient le pattern PROXY sont parfois fragiles car elles s'appuient sur la transmission d'appels de méthodes à des objets sous-jacents. Cette transmission peut produire une conception fragile et coûteuse en maintenance.

Imaginez que vous soyez ingénieur chez Oozinoz et travailliez à un proxy d'image qui, pour des raisons de performances, affichera une petite image temporaire pendant le chargement d'une image plus volumineuse. Vous disposez d'un prototype opérationnel (voir Figure 11.1). Le code de cette application est contenu dans la classe ShowProxy du package app.proxy. Le code sous-jacent qui supporte cette application se trouve dans le package com.oozinoz.imaging.

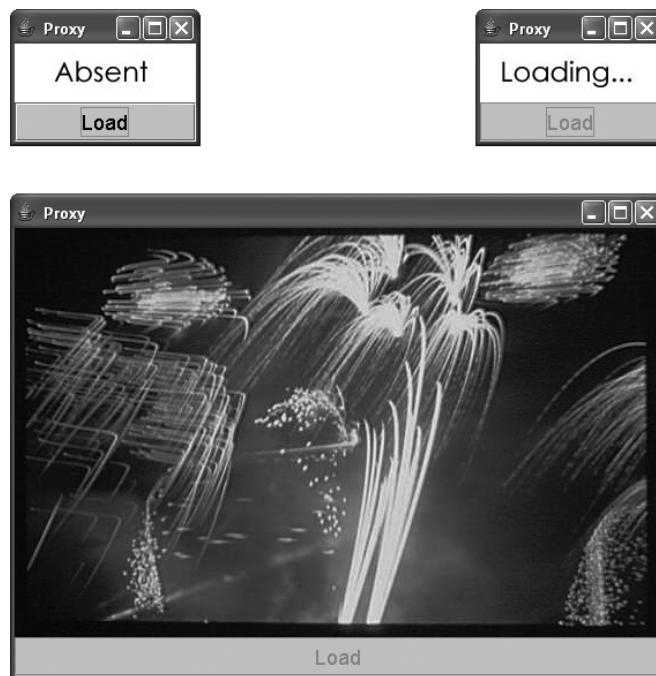


Figure 11.1

Les captures d'écran illustrent une mini-application avant, pendant et après le chargement d'une image volumineuse (cette image appartient au domaine public. Library of Congress, Prints and Photographs Division, Gottscho-Schleisner Collection [LC-G605-CT-00488]).

L'interface utilisateur affiche l'une des trois images suivantes : une image indiquant que le chargement n'a pas encore commencé (Absent), une image indiquant que l'image est en cours de chargement (Loading...), ou l'image voulue. Lorsque l'application démarre, elle affiche Absent, une image JPEG créée à l'aide d'un outil de dessin. Lorsque l'utilisateur clique sur Load, une image Loading... prédefinie s'affiche presque instantanément. Après quelques instants, l'image désirée apparaît.

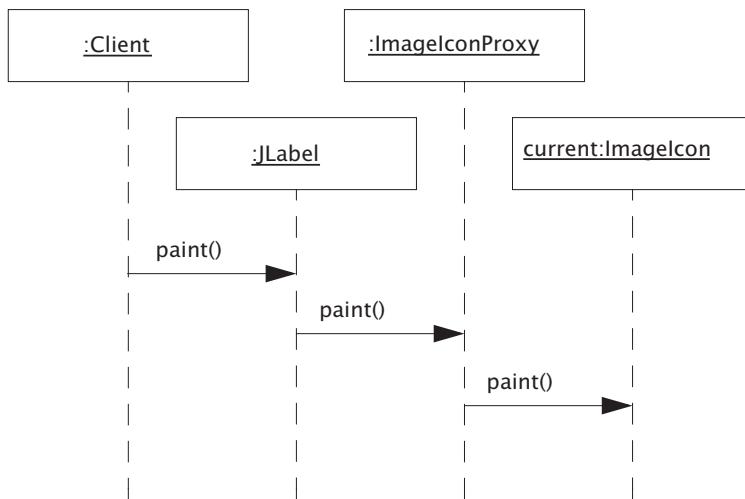
Un moyen ais\u00e9 d'afficher une image enregistr\u00e9e au format JPEG, par exemple, est d'utiliser un objet `ImageIcon` comme argument d'un "label" qui affichera l'image :

```
 ImageIcon icon = new ImageIcon("images/fest.jpg");
 JLabel label = new JLabel(icon);
```

Dans l'application que vous d\u00e9veloppez, vous voulez passer \u00e0 `JLabel` un proxy qui transmettra les requ\u00eetes de dessin de l'\u00e9cran (*paint*) \u00e0 : (1) une image `Absent`, (2) une image `Loading...`, ou (3) l'image d\u00e9sir\u00e9e. Le flux des messages est repr\u00e9sent\u00e9 dans le diagramme de s\u00e9quence de la Figure 11.2.

Figure 11.2

Un objet `ImageIconProxy` transmet les requ\u00eetes `paint()` \u00e0 l'objet `ImageIcon` courant.

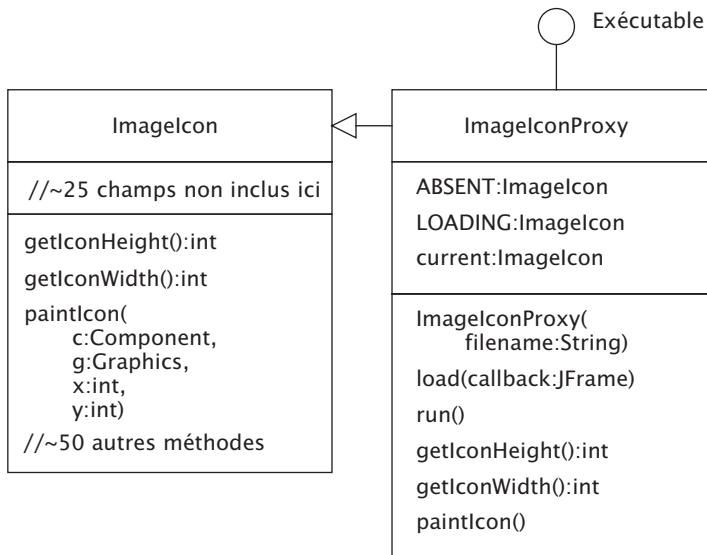


Lorsque l'utilisateur clique sur Load, votre code fait en sorte que l'image courante de l'objet `ImageIconProxy` devienne `Loading...`, et le proxy entame le chargement de l'image attendue. Une fois celle-ci compl\u00e8tement charg\u00e9e, elle devient l'image courante de `ImageIconProxy`.

Pour cr\u00e9er un proxy, vous pouvez d\u00e9river une sous-classe de `ImageIcon`, comme le montre la Figure 11.3. Le code de `ImageIconProxy` d\u00e9finit deux variables statiques contenant les images `Absent` et `Loading...` :

```
 static final ImageIcon ABSENT = new ImageIcon(
    ClassLoader.getSystemResource("images/absent.jpg"));

 static final ImageIcon LOADING = new ImageIcon(
    ClassLoader.getSystemResource("images/loading.jpg"));
```

**Figure 11.3**

Un objet `ImageIconProxy` peut remplacer un objet `ImageIcon` puisqu'il s'agit en fait d'un objet `ImageIcon`.

Le constructeur de `ImageIconProxy` reçoit le nom d'un fichier d'image à charger. Lorsque la méthode `load()` d'un objet `ImageIconProxy` est invoquée, elle définit l'image comme étant `LOADING` et lance un thread séparé pour charger l'image. Le fait d'employer un thread séparé évite à l'application de devoir patienter pendant le chargement. La méthode `load()` reçoit un objet `JFrame` qui est rappelé par la méthode `run()` à l'issue du chargement. Voici le code presque complet de `ImageIconProxy.java` :

```

package com.oozinoz.imaging;
import java.awt.*;
import javax.swing.*;

public class ImageIconProxy
    extends ImageIcon implements Runnable {
    static final ImageIcon ABSENT = new ImageIcon(
        ClassLoader.getSystemResource("images/absent.jpg"));
    static final ImageIcon LOADING = new ImageIcon(
        ClassLoader.getSystemResource("images/loading.jpg"));
    ImageIcon current = ABSENT;
    protected String filename;
    protected JFrame callbackFrame;

    public ImageIconProxy(String filename) {

```

```
super(ABSENT.getImage());
this.filename = filename;
}

public void load(JFrame callbackFrame) {
    this.callbackFrame = callbackFrame;
    current = LOADING;
    callbackFrame.repaint();
    new Thread(this).start();
}

public void run() {
    current = new ImageIcon(
        ClassLoader.getSystemResource(filename));
    callbackFrame.pack();
}

public int getIconHeight() { /* Exercice ! */ }

public int getIconWidth() { /* Exercice ! */ }

public synchronized void paintIcon(
    Component c, Graphics g, int x, int y) {
    // Exercice !
}
}
```

Exercice 11.1

Un objet `ImageIconProxy` accepte trois appels d'affichage d'image qu'il doit passer à l'image courante. Ecrivez le code des méthodes `getIconHeight()`, `getIconWidth()` et `paintIcon()` de la classe `ImageIconProxy`.

▪ *Les solutions des exercices de ce chapitre sont données dans l'Annexe B.*

Imaginez que vous parveniez à faire fonctionner le code de cette petite application de démonstration. Avant de créer la véritable application, laquelle ne se limite pas à un bouton Load, vous procédez à une révision de la conception, qui révèle toute sa fragilité.

Exercice 11.2

La classe `ImageIconProxy` ne constitue pas un composant réutilisable bien conçu. Citez deux problèmes de cette conception.

Lorsque vous révisez la conception d'un développeur, vous devez à la fois comprendre cette conception et vous former une opinion à son sujet. Il se peut que le développeur pense avoir utilisé un pattern spécifique alors que vous doutez qu'il soit présent. Dans l'exemple précédent, le pattern PROXY apparaît de manière évidente mais ne garantit en rien que la conception soit bonne. Il existe d'ailleurs de bien meilleures conceptions. Lorsque ce pattern est présent, il doit pouvoir être justifié car la transmission de requêtes peut entraîner des problèmes que d'autres conceptions permettraient d'éviter. La prochaine section devrait vous aider à déterminer si le pattern PROXY est une option valable pour votre conception.

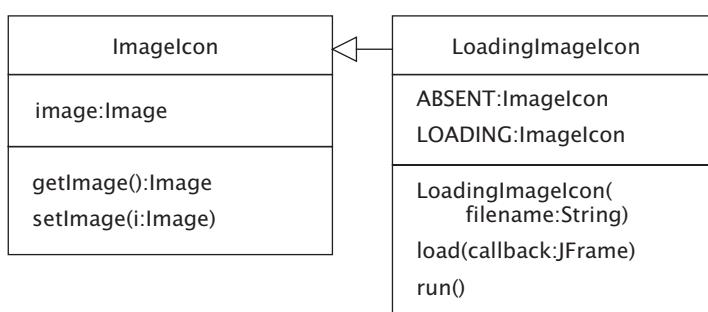
Reconsidération des proxies d'image

A ce stade, peut-être vous demandez-vous si les patterns de conception vous ont été d'une quelconque aide. Vous avez implémenté fidèlement un pattern et voilà que vous cherchez maintenant à vous en débarrasser. Il s'agit en fait d'une étape naturelle et même saine, qui survient plus souvent dans des conditions réelles de développement que dans les livres. En effet, un auteur peut, avec l'aide de ses relecteurs, repenser et remplacer une conception de qualité insuffisante avant que son ouvrage ne soit publié. Dans la pratique, un pattern peut vous aider à faire fonctionner une application et faciliter les discussions sur sa conception. Dans l'exemple de `ImageIconProxy`, le pattern a servi à cela, même s'il est beaucoup plus simple d'obtenir l'effet désiré sans implémenter littéralement un proxy.

La classe `ImageIcon` opère sur un objet `Image`. Plutôt que de transmettre les requêtes de dessin de l'écran à un objet `ImageIcon` séparé, il est plus facile d'opérer sur l'objet `Image` enveloppé dans `ImageIcon`. La Figure 11.4 présente une classe `LoadingImageIcon` (tirée du package `com.oozinoz.imaging`) qui possède seulement deux méthodes, `load()` et `run()`, en plus de ses constructeurs.

Figure 11.4

La classe `LoadingImageIcon` fonctionne en changeant l'objet `Image` qu'elle contient.



La méthode `load()` de cette classe révisée reçoit toujours un objet `JFrame` à rappeler après le chargement de l'image souhaitée. Lorsqu'elle s'exécute, elle invoque `setImage()` avec l'image de `LOADING`, redessine le cadre (*frame*) et lance un thread séparé pour elle-même. La méthode `run()`, qui s'exécute dans un thread séparé, crée un nouvel objet `ImageIcon` pour le fichier nommé dans le constructeur, appelle `setImage()` avec l'image de cet objet et redessine le cadre.

Voici le code presque complet de `LoadingImageIcon.java` :

```
package com.oozinoz.imaging;
import javax.swing.ImageIcon;
import javax.swing.JFrame;

public class LoadingImageIcon
    extends ImageIcon implements Runnable {
    static final ImageIcon ABSENT = new ImageIcon(
        ClassLoader.getSystemResource("images/absent.jpg"));
    static final ImageIcon LOADING = new ImageIcon(
        ClassLoader.getSystemResource("images/loading.jpg"));
    protected String filename;
    protected JFrame callbackFrame;

    public LoadingImageIcon(String filename) {
        super(ABSENT.getImage());
        this.filename = filename;
    }

    public void load(JFrame callbackFrame) {
        // Exercice !
    }

    public void run() {
        // Exercice !
    }
}
```

Exercice 11.3

Ecrivez le code des méthodes `load()` et `run()` de `LoadingImageIcon`.

Ce code révisé est moins lié à la conception de `ImageIcon`, s'appuyant principalement sur `getImage()` et `setImage()` et non sur la transmission d'appels. En fait, il n'y a pas du tout de transmission. `LoadingImageIcon` a seulement l'apparence d'un proxy, et non la structure.

Le fait que le pattern PROXY ait recours à la transmission peut accroître la maintenance du code. Par exemple, si l'objet sous-jacent change, l'équipe d'Oozinoz devra actualiser le proxy. Pour éviter cela, vous devriez lorsque vous le pouvez renoncer à ce pattern. Il existe cependant des situations où vous n'avez d'autre choix que de l'utiliser. En particulier, lorsque l'objet pour lequel vous devez intercepter des messages s'exécute sur une autre machine, ce pattern est parfois la seule option envisageable.

Proxy distant

Lorsque vous voulez invoquer une méthode d'un objet qui s'exécute sur un autre ordinateur, vous ne pouvez le faire directement et devez donc trouver un autre moyen de communiquer avec lui. Vous pourriez ouvrir un socket sur l'hôte distant et élaborer un protocole pour envoyer des messages à l'objet. Idéalement, une telle approche vous permettrait de lui passer des messages de la même manière que s'il était local. Vous devriez pouvoir appeler les méthodes d'un objet proxy qui transmettrait ces requêtes à l'objet distant. En fait, de telles conceptions ont déjà été implémentées, notamment dans **CORBA** (*Common Object Request Broker Architecture*), dans ASP.NET (*Active Server Pages for .NET*), et dans Java **RMI** (*Remote Method Invocation*).

Grâce à RMI, un client peut assez aisément obtenir un objet proxy qui transmette les appels vers l'objet désiré actif sur une autre machine. Il importe de connaître RMI puisque ce mécanisme fait partie des fondements de la spécification **EJB** (*Enterprise JavaBeans*), un standard important de l'industrie. Indépendamment de la façon dont les standards de l'industrie évoluent, le pattern PROXY continuera de jouer un rôle important dans les environnements distribués, du moins dans un avenir proche, et RMI représente un bon exemple d'implémentation de ce pattern.

Pour vous familiariser avec RMI, vous aurez besoin d'un ouvrage de référence sur le sujet, tel que *Java™ Enterprise in a Nutshell* (*Java en concentré : Manuel de référence pour Java*) [Flanagan et al. 2002]. L'exemple présenté dans cette section n'est pas un tutoriel sur RMI mais permet de mettre en évidence la présence et l'importance du pattern PROXY dans les applications RMI. Nous laisserons de côté les difficultés de conception introduites par RMI et EJB.

Supposez que vous ayez décidé d'explorer le fonctionnement de RMI en rendant les méthodes d'un objet accessibles à un programme Java qui s'exécute sur un autre

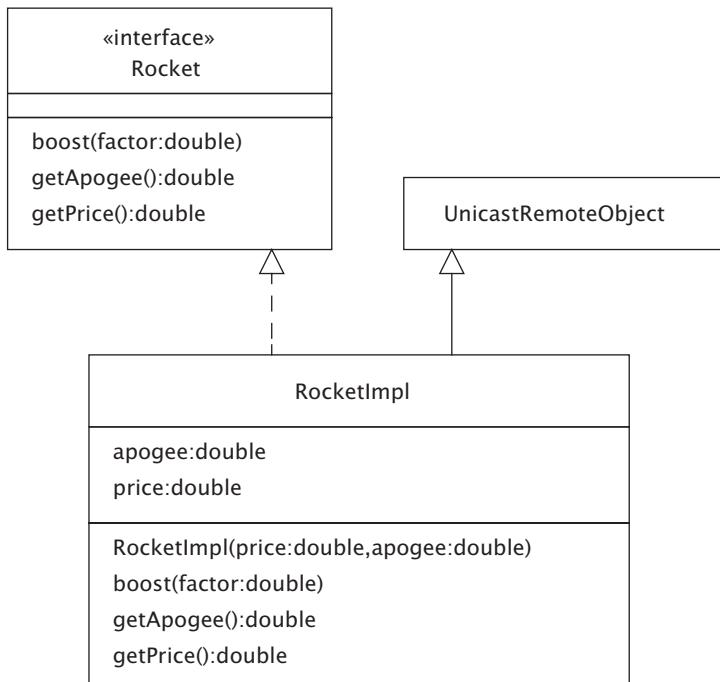
ordinateur. La première étape de développement consiste à créer une interface pour la classe qui doit être accessible à distance. Vous commencez par créer une interface Rocket qui est indépendante du code existant à Oozinoz :

```
package com.oozinoz.remote;
import java.rmi.*;
public interface Rocket extends Remote {
    void boost(double factor) throws RemoteException;
    double getApogee() throws RemoteException;
    double getPrice() throws RemoteException;
}
```

L'interface Rocket étend Remote et ses méthodes déclarent toutes qu'elles génèrent (*throw*) des exceptions distantes (RemoteException). Expliquer cet aspect de l'interface dépasse le cadre du présent livre, mais n'importe quel ouvrage didacticiel sur RMI devrait le faire. Votre référence RMI devrait également expliquer que, pour agir en tant que serveur, l'implémentation de votre interface distante peut étendre UnicastRemoteObject, comme illustré Figure 11.5.

Figure 11.5

Pour utiliser RMI, vous pouvez d'abord définir l'interface souhaitée pour les messages échangés entre les deux ordinateurs puis créer une sous-classe de UnicastRemoteObject qui implémente cette interface.



Vous avez prévu que des objets RocketImpl soient actifs sur un serveur et accessibles via un proxy qui lui est actif sur un client. Le code de la classe RocketImpl est simple :

```
package com.oozinoz.remote;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class RocketImpl
    extends UnicastRemoteObject
    implements Rocket {
    protected double price;
    protected double apogee;

    public RocketImpl(double price, double apogee)
        throws RemoteException {
        this.price = price;
        this.apogee = apogee;
    }

    public void boost(double factor) {
        apogee *= factor;
    }

    public double getApogee() {
        return apogee;
    }

    public double getPrice() {
        return price;
    }
}
```

Une instance de RocketImpl peut être active sur une machine et accessible à un programme Java exécuté sur une autre machine. Pour que cela puisse fonctionner, un client a besoin d'un proxy pour l'objet RocketImpl. Ce proxy doit implémenter l'interface Rocket et posséder les fonctionnalités additionnelles requises pour communiquer avec un objet distant. Un gros avantage de RMI est qu'il automatise la construction de ce proxy. Pour générer le proxy, placez le fichier Rocket-Impl.java et le fichier d'interface Rocket.java sous le répertoire dans lequel vous exécuterez le registre RMI :

```
c:\rmi>dir /b com\oozinoz\remote
RegisterRocket.class
RegisterRocket.java
Rocket.class
```

```
Rocket.java  
RocketImpl.class  
RocketImpl.java  
ShowRocketClient.class  
ShowRocketClient.java
```

Pour créer la classe stub `RocketImpl` qui facilite la communication distante, exécutez le compilateur RMI livré avec le JDK :

```
c:\rmi> rmic com.oozinoz.remote.RocketImpl
```

Notez que l'exécutable `rmic` reçoit comme argument un nom de classe, et non un nom de fichier. Depuis la version 1.2 du JDK, le compilateur RMI crée un seul fichier stub dont les machines client et serveur ont toutes deux besoin. Les versions antérieures créaient des fichiers séparés pour une utilisation sur le client et le serveur. La commande `rmic` crée une classe `RocketImpl_Stub` :

```
c:\rmi>dir /b com\oozinoz\remote  
RegisterRocket.class  
RegisterRocket.java  
Rocket.class  
Rocket.java  
RocketImpl.class  
RocketImpl.java  
RocketImpl_Stub.class  
ShowRocketClient.class  
ShowRocketClient.java
```

Pour rendre un objet actif, il faut l'enregistrer auprès d'un registre RMI qui s'exécute sur le serveur. L'exécutable `rmiregistry` est intégré au JDK. Lorsque vous exécutez le registre, spécifiez le port sur lequel il écoutera :

```
c:\rmi> rmiregistry 5000
```

Une fois le registre en cours d'exécution sur le serveur, vous pouvez créer et enregistrer un objet `RocketImpl` :

```
package com.oozinoz.remote;  
import java.rmi.*;  
public class RegisterRocket {  
    public static void main(String[] args) {  
        try {  
            // Exercice !  
            Naming.rebind(  
                "rmi://localhost:5000/Biggie", biggie);  
            System.out.println("biggie enregistré");  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Si vous compilez et exédez ce code, le programme affichera une confirmation de l'enregistrement de la fusée :

```
biggie enregistré
```

Vous devez remplacer la ligne `// Exercice !` de la classe `RegisterRocket` par le code qui crée un objet `biggie` modélisant une fusée. Le reste du code de la méthode `main()` enregistre cet objet. Une description du fonctionnement de la classe `Naming` dépasse le cadre de cette discussion. Vous devriez néanmoins disposer des informations suffisantes pour pouvoir créer l'objet `biggie` que ce code enregistre.

Exercice 11.4

Remplacez la ligne `// Exercice !` par une déclaration et une instanciation de l'objet `biggie`. Définissez cet objet de sorte qu'il modélise une fusée dont le prix est de 29,95 dollars et l'apogée de 820 mètres.

Le fait d'exécuter le programme `RegisterRocket` rend un objet `RocketImpl`, en l'occurrence `biggie`, disponible sur un serveur. Un client qui s'exécute sur une autre machine peut accéder à `biggie` s'il dispose d'un accès à l'interface `Rocket` et à la classe `RocketImpl_Stub`. Si vous travaillez sur une seule machine, vous pouvez quand même réaliser ce test en accédant au serveur sur `localhost` plutôt que sur un autre hôte :

```
package com.oozinoz.remote;

import java.rmi.*;
public class ShowRocketClient {
    public static void main(String[] args) {
        try {
            Object obj = Naming.lookup(
                "rmi://localhost:5000/Biggie");
            Rocket biggie = (Rocket) obj;
            System.out.println(
                "L'apogée est " + biggie.getApogee());
```

```
        } catch (Exception e) {
            System.out.println(
                "Exception lors de la recherche d'une fusée :");
            e.printStackTrace();
        }
    }
}
```

Lorsque ce programme s'exécute, il recherche un objet enregistré sous le nom de "Biggie". La classe qui fournit ce nom est RocketImpl et l'objet obj retourné par lookup() sera une instance de la classe RocketImpl_Stub. Cette dernière implémente l'interface Rocket, aussi est-il légal de convertir (*cast*) l'objet obj en une instance de Rocket. La classe RocketImpl_Stub étend en fait une classe RemoteStub qui permet à l'objet de communiquer avec un serveur.

Lorsque vous exécutez le programme ShowRocketClient, il affiche l'apogée d'une fusée "Biggie" :

```
L'apogée est de 820.0
```

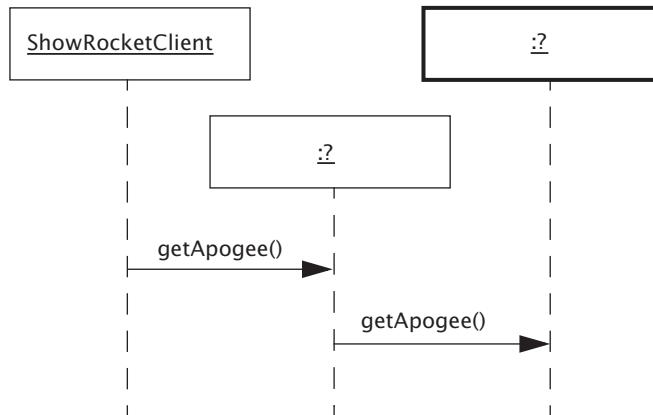
Par l'intermédiaire d'un proxy, l'appel de getApogee() est transmis à une implémentation de l'interface Rocket qui est active sur un serveur.

Exercice 11.5

La Figure 11.6 illustre l'appel de getApogee() qui est transmis. L'objet le plus à droite apparaît en gras pour signifier qu'il est actif en dehors du programme ShowRocketClient. Complétez les noms de classes manquants.

Figure 11.6

Ce diagramme, une fois complété, représentera le flux de messages dans une application distribuée basée sur RMI.



L'intérêt de RMI est qu'il permet à des programmes client d'interagir avec un objet local servant de proxy pour un objet distant. Vous définissez l'interface de l'objet que sont censés se partager le client et le serveur. RMI fournit, lui, le mécanisme de communication et dissimule au serveur et au client le fait que deux implémentations de Rocket collaborent pour assurer une communication interprocessus quasiment transparente.

Proxy dynamique

Les ingénieurs d'Oozinoz sont parfois confrontés à des problèmes de performances et aimeraient trouver un moyen d'instrumentaliser le code sans avoir à apporter de changements majeurs à leur conception.

Java offre une fonctionnalité qui peut les aider dans ce sens : le **proxy dynamique**. Un tel proxy permet d'envelopper un objet dans un autre. Vous pouvez faire en sorte que l'objet extérieur — le proxy — intercepte tous les appels destinés à l'objet enveloppé. Le proxy passe habituellement ces appels à l'objet intérieur, mais vous pouvez ajouter du code qui s'exécute avant ou après les appels interceptés. Certaines limitations de cette fonctionnalité vous empêchent d'envelopper n'importe quel objet. En revanche, dans des conditions adéquates, vous disposez d'un contrôle total sur l'opération accomplie par l'objet enveloppé.

Un proxy dynamique utilise les *interfaces* implementées par la classe d'un objet. Les appels pouvant être interceptés par le proxy sont définis dans l'une de ces interfaces. Si vous disposez d'une classe qui implemente une interface dont vous voulez intercepter certaines méthodes, vous pouvez utiliser un proxy dynamique pour envelopper une instance de cette classe.

Pour créer un proxy dynamique, vous devez avoir la liste des interfaces à intercepter. Heureusement, cette liste peut généralement être obtenue en interrogeant l'objet que vous voulez envelopper au moyen d'une ligne de code comme la suivante :

```
Class[] classes = obj.getClass().getInterfaces();
```

Ce code indique que les méthodes à intercepter appartiennent aux interfaces implementées par la classe d'un objet. La création d'un proxy dynamique nécessite deux autres ingrédients : un chargeur de classe (*loader*) et une classe contenant le comportement que vous voulez exécuter lorsque votre proxy intercepte un appel.

Comme pour la liste d'interfaces, vous pouvez obtenir un chargeur de classe approprié en utilisant celui associé à l'objet à envelopper :

```
ClassLoader loader = obj.getClass().getClassLoader();
```

Le dernier ingrédient requis est l'objet proxy lui-même. Cet objet doit être une instance d'une classe qui implémente l'interface `InvocationHandler` du package `java.lang.reflect`. Cette interface déclare l'opération suivante :

```
public Object invoke(Object proxy, Method m, Object[] args)
    throws Throwable;
```

Lorsque vous enveloppez un objet dans un proxy dynamique, les appels destinés à cet objet sont déroutés vers cette opération `invoke()`, dans une classe que vous fournissez. Le code de votre méthode `invoke()` devra probablement passer à l'objet enveloppé chaque appel de méthode. Vous pouvez passer l'invocation avec une ligne comme celle-ci :

```
result = m.invoke(obj, args);
```

Cette ligne utilise le principe de réflexion pour passer l'appel désiré à l'objet enveloppé. Le grand intérêt des proxies dynamiques est que vous pouvez ajouter n'importe quel comportement avant ou après l'exécution de cette ligne.

Imaginez que vous vouliez consigner un avertissement lorsqu'une méthode est longue à s'exécuter. Vous pourriez créer une classe `ImpatientProxy` avec le code suivant :

```
package app.proxy.dynamic;

import java.lang.reflect.*;

public class ImpatientProxy implements InvocationHandler {
    private Object obj;

    private ImpatientProxy(Object obj) {
        this.obj = obj;
    }

    public Object invoke(
        Object proxy, Method m, Object[] args)
        throws Throwable {
        Object result;
        long t1 = System.currentTimeMillis();
        result = m.invoke(obj, args);
        long t2 = System.currentTimeMillis();
        if (t2 - t1 > 1000) {
            System.out.println("Warning: " + m.getName() +
                " took " + (t2 - t1) + " ms");
        }
        return result;
    }
}
```

```

        long t2 = System.currentTimeMillis();
        if (t2 - t1 > 10) {
            System.out.println(
                "> Il faut " + (t2 - t1)
                + " millisecondes pour invoquer " + m.getName()
                + "() avec");
            for (int i = 0; i < args.length; i++)
                System.out.println(
                    "> arg[" + i + "]: " + args[i]);
        }
        return result;
    }
}

```

Cette classe implémente la méthode `invoke()` de manière qu'elle vérifie le temps que prend l'objet enveloppé pour accomplir l'opération invoquée. Si la durée d'exécution est trop longue, la classe `ImpatientProxy` affiche un avertissement.

Pour pouvoir utiliser un objet `ImpatientProxy`, vous devez employer la classe `Proxy` du package `java.lang.reflect`. Cette classe a besoin d'une liste d'interfaces et d'un chargeur de classe, ainsi que d'une instance de `ImpatientProxy`. Pour simplifier la création du proxy dynamique, nous pourrions ajouter la méthode suivante à la classe `ImpatientProxy` :

```

public static Object newInstance(Object obj) {
    ClassLoader loader = obj.getClass().getClassLoader();
    Class[] classes = obj.getClass().getInterfaces();
    return Proxy.newProxyInstance(
        loader, classes, new ImpatientProxy(obj));
}

```

La méthode statique `newInstance()` crée un proxy dynamique pour nous. A partir d'un objet à envelopper, elle extrait la liste des interfaces et le chargeur de classe de cet objet. Puis elle instancie la classe `ImpatientProxy` en lui passant l'objet à envelopper. Tous ces ingrédients sont ensuite passés à la méthode `newProxyInstance()` de la classe `Proxy`.

L'objet retourné implémente toutes les interfaces implémentées par la classe de l'objet enveloppé. Nous pouvons convertir l'objet retourné en n'importe laquelle de ces interfaces.

Imaginez que vous travailliez avec un ensemble (*set*) d'objets et que certaines opérations semblent s'exécuter lentement. Pour déterminer quels objets sont

concernés, vous pouvez envelopper l'ensemble dans un objet `ImpatientProxy`, comme illustré ci-après :

```
package app.proxy.dynamic;

import java.util.HashSet;
import java.util.Set;
import com.oozinoz.firework.Firecracker;
import com.oozinoz.firework.Sparkler;
import com.oozinoz.utility.Dollars;

public class ShowDynamicProxy {
    public static void main(String[] args) {
        Set s = new HashSet();
        s = (Set)ImpatientProxy.newInstance(s);
        s.add(new Sparkler(
            "Mr. Twinkle", new Dollars(0.05)));
        s.add(new BadApple("Lemon"));
        s.add(new Firecracker(
            "Mr. Boomy", new Dollars(0.25)));
        System.out.println(
            "L'ensemble contient " + s.size() + " éléments.");
    }
}
```

Ce code crée un objet `Set` pour contenir quelques éléments. Puis il enveloppe cet ensemble dans un objet `ImpatientProxy`, convertissant le résultat de la méthode `newInstance()` en un objet `Set`. La conséquence est que l'objet `s` se comporte comme un ensemble, sauf que le code de `ImpatientProxy` produira un avertissement si une des méthodes est trop longue à s'exécuter. Par exemple, lorsque le programme invoque la méthode `add()` de l'ensemble, notre objet `ImpatientProxy` intercepte l'appel et le passe à l'ensemble en minutant le résultat de chaque appel.

L'exécution du programme `ShowDynamicProxy` produit le résultat suivant :

```
> Il faut 1204 millisecondes pour invoquer add() avec
> arg[0]: Lemon
L'ensemble contient 3 éléments.
```

Le code de `ImpatientProxy` nous aide à identifier l'objet qui est long à ajouter à l'ensemble. Il s'agit de l'instance `Lemon` de la classe `BadApple`. Voici le code de cette classe :

```
package app.proxy.dynamic;

public class BadApple {
    public String name;
```

```
public BadApple(String name) {
    this.name = name;
}

public boolean equals(Object o) {
    if (!(o instanceof BadApple))
        return false;
    BadApple f = (BadApple) o;
    return name.equals(f.name);
}

public int hashCode() {
    try {
        Thread.sleep(1200);
    } catch (InterruptedException ignored) {
    }
    return name.hashCode();
}

public String toString() {
    return name;
}
}
```

Le code de `ShowDynamicProxy` utilise un objet `ImpatientProxy` pour surveiller les appels destinés à un ensemble. Il n'existe toutefois aucun lien entre un ensemble donné et `ImpatientProxy`. Après avoir écrit une classe de proxy dynamique, vous pouvez l'utiliser pour envelopper n'importe quel objet dès lors que celui-ci est une instance d'une classe qui implémente une interface déclarant le comportement que vous voulez intercepter.

La possibilité de créer un comportement pouvant être exécuté avant ou après les appels interceptés est l'une des idées de base de la programmation orientée aspect ou POA (*Aspect-Oriented Programming*). Un *aspect* combine les notions d'*advice* — le code que vous voulez insérer — et de *point-cuts* — la définition de points d'exécution où vous voulez que le code inséré soit exécuté. Des livres entiers sont consacrés à la POA, mais vous pouvez avoir un avant-goût de l'application de comportements réutilisables à une variété d'objets en utilisant des proxies dynamiques.

Un proxy dynamique vous permet d'envelopper un objet dans un proxy qui intercepte les appels destinés à cet objet et qui ajoute un comportement avant ou après le passage de ces appels à l'objet enveloppé. Vous pouvez ainsi créer des comportements réutilisables applicables à n'importe quel objet, comme en programmation orientée aspect.

Résumé

Les implémentations du pattern PROXY produisent un objet intermédiaire qui gère l'accès à un objet cible. Un objet proxy peut dissimuler aux clients les changements d'état d'un objet cible, comme dans le cas d'une image qui nécessite un certain temps pour se charger. Le problème est que ce pattern s'appuie habituellement sur un couplage étroit entre l'intermédiaire et l'objet cible. Dans certains cas, la solution consiste à utiliser un proxy dynamique. Lorsque la classe d'un objet implémente des interfaces pour les méthodes que vous voulez intercepter, vous pouvez envelopper l'objet dans un proxy dynamique et faire en sorte que votre code s'exécute avant/après le code de l'objet enveloppé ou à sa place.

CHAIN OF RESPONSABILITY

Les développeurs s'efforcent d'associer les objets de manière souple avec une responsabilité minimale et spécifique entre objets. Cette pratique permet de procéder plus facilement à des changements et avec moins de risques d'introduire des défauts. Dans une certaine mesure, la dissociation se produit naturellement en Java. Les clients ne voient que l'interface visible d'un objet et sont affranchis des détails de son implémentation. Cette organisation laisse toutefois en place l'association fondamentale pour que le client sache quel objet possède la méthode qu'il doit appeler. Vous pouvez assouplir la restriction forçant un client à savoir quel objet utiliser lorsque vous pouvez organiser un groupe d'objets sous forme d'une sorte de hiérarchie qui permet à chaque objet soit de réaliser une opération, soit de passer la requête à un autre objet.

L'objectif du pattern CHAIN OF RESPONSABILITY est d'éviter de coupler l'émetteur d'une requête à son récepteur en permettant à plus d'un objet d'y répondre.

Une chaîne de responsabilités ordinaires

Le pattern CHAIN OF RESPONSABILITY apparaît souvent dans notre quotidien réel lorsqu'une personne responsable d'une tâche s'en acquitte personnellement ou la délègue à quelqu'un d'autre. Cette situation se produit chez Oozinoz avec des ingénieurs responsables de la maintenance des machines de fabrication de fusées.

Comme décrit au Chapitre 5, Oozinoz modélise des machines, des lignes de montage, des travées, et des unités de production (ou usine) en tant que composants matériels de fabrication (objet `MachineComponent`). Cette approche permet l'implémentation

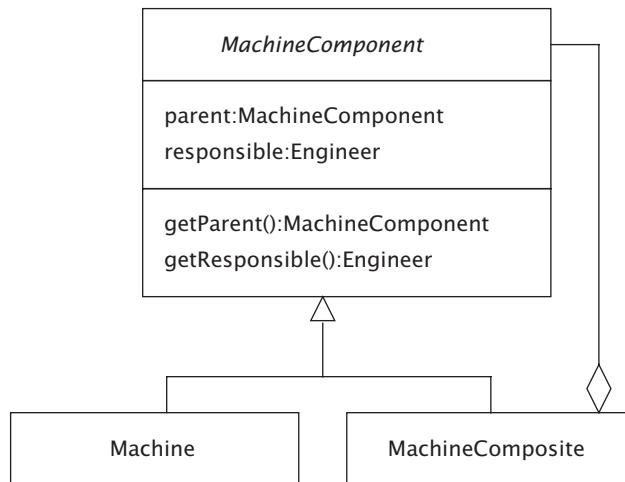
simple et récursive d'opérations telles que l'arrêt de toutes les machines d'une travée. Elle simplifie aussi la modélisation des responsabilités de fabrication au sein de l'usine. Chez Oozinoz, il y a toujours un ingénieur responsable pour n'importe quel composant matériel, bien que cette responsabilité puisse être assignée à différents niveaux.

Par exemple, il peut y avoir un ingénieur directement assigné à la maintenance d'une machine complexe mais pas forcément dans le cas d'une machine simple. Dans ce dernier cas, c'est l'ingénieur responsable de la ligne ou de la travée à laquelle participe la machine qui en assumera la responsabilité.

Nous voudrions ne pas forcer les objets clients à interroger plusieurs objets lorsqu'ils recherchent l'ingénieur responsable. Nous pouvons ici appliquer le pattern CHAIN OF RESPONSABILITY, en associant à chaque composant matériel un objet responsable. La Figure 12.1 illustre cette conception.

Figure 12.1

Chaque objet Machine Machine-Composite possède un parent et une association de responsabilité, hérités de la classe MachineComponent.



La conception illustrée Figure 12.1 permet, sans qu'on ait à le requérir, que chaque composant matériel garde trace de son ingénieur responsable. Si une machine n'a pas d'ingénieur dédié, elle peut passer une requête demandant à son ingénieur responsable d'être son "parent". Dans la pratique, le parent d'une machine est une ligne, celui d'une ligne est une travée, et celui d'une travée est une unité de production. Chez Oozinoz, il y a toujours un ingénieur responsable quelque part dans cette chaîne.

L'avantage de cette conception est que les clients de composants matériels n'ont pas besoin de déterminer comment les ingénieurs sont attribués. Un client peut demander à n'importe quel composant son ingénieur responsable. Les composants évitent aux clients d'avoir à connaître la façon dont les responsabilités sont distribuées. D'un autre côté, cette conception présente quelques éventuels inconvénients.

Exercice 12.1

Citez deux faiblesses de la conception illustrée Figure 12.1.

- *Les solutions des exercices de ce chapitre sont données dans l'Annexe B.*

Le pattern CHAIN OF RESPONSABILITY permet de simplifier le code du client lorsqu'il n'est pas évident de savoir quel objet d'un groupe d'objets doit traiter une requête. Si ce pattern n'était pas déjà implémenté, vous pourriez remarquer certaines situations où il pourrait vous aider à migrer votre code vers une conception plus simple.

Refactorisation pour appliquer CHAIN OF RESPONSABILITY

Si vous remarquez qu'un code client effectue des appels de test avant d'émettre la requête effective, vous pourriez l'améliorer au moyen d'une refactorisation. Pour appliquer le pattern CHAIN OF RESPONSABILITY, déterminez l'opération que les objets d'un groupe de classes seront parfois en mesure de supporter. Par exemple, les composants matériels chez Oozinoz peuvent parfois fournir une référence d'ingénieur responsable. Ajoutez l'opération souhaitée à chaque classe dans le groupe, mais implémentez l'opération au moyen d'une stratégie de chaînage pour les cas où un objet spécifique nécessiterait de l'aide pour répondre à la requête.

Considérez le code Oozinoz de modélisation d'outils (*Tool*) et de chariots d'outils (*Tool Cart*). Les outils ne font pas partie de la hiérarchie *MachineComponent* mais ils partagent quelques similitudes avec ces composants. Plus précisément, les outils sont toujours assignés aux chariots d'outils, et ces derniers ont un ingénieur responsable. Imaginez un affichage pouvant montrer tous les outils et les machines d'une certaine travée et disposant d'une aide affichant l'ingénieur responsable pour

n’importe quel élément choisi. La Figure 12.2 illustre les classes impliquées dans l’identification de l’ingénieur responsable d’un équipement sélectionné.

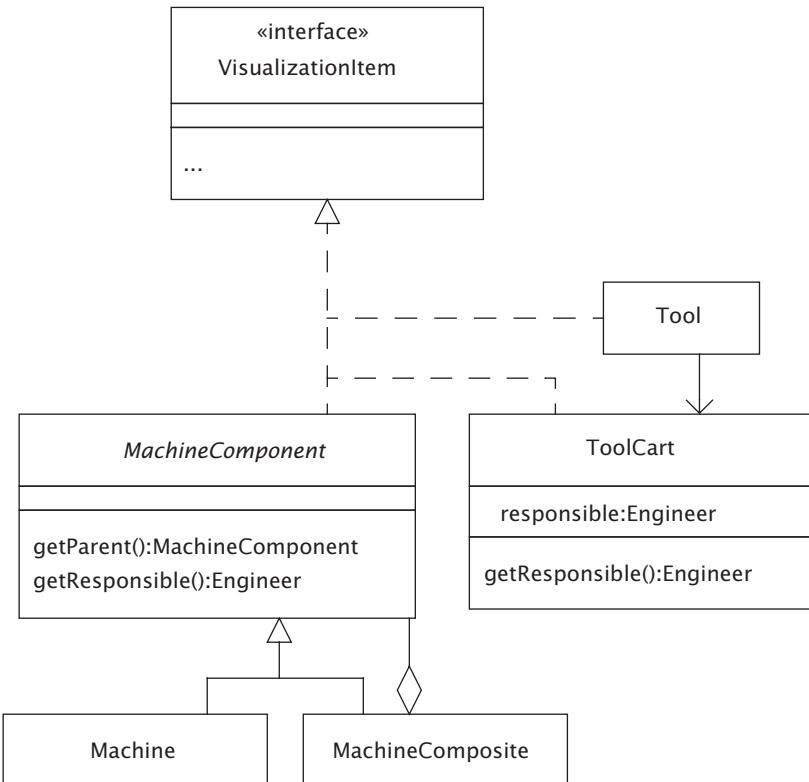


Figure 12.2

Les éléments d’une simulation comprennent des machines, des machines composites, des outils et des chariots d’outils.

L’interface `VisualizationItem` spécifie quelques comportements que les classes requièrent pour participer à l’affichage, mais ne possède pas de méthode `getResponsible()`. En fait, tous les éléments de la visualisation n’ont pas une connaissance directe de leur responsable. Lorsque la visualisation doit déterminer l’ingénieur responsable d’un élément, la réponse dépend du type de l’élément sélectionné. Les machines, les groupes de machines et les chariots d’outils disposent d’une méthode `getResponsible()`, mais pas les outils. Pour ceux-ci, le code doit

identifier le chariot auquel appartient l'outil et déterminer le responsable du chariot. Pour trouver l'ingénieur responsable d'un élément simulé, un code de menu d'application utilise une série d'instructions `if` et de tests du type d'élément. Cela est le signe qu'une refactorisation pourrait améliorer le code qui se présente comme suit :

```
package com.oozinoz.machine;

public class AmbitiousMenu {
    public Engineer getResponsible(VisualizationItem item) {
        if (item instanceof Tool) {
            Tool t = (Tool) item;
            return t.getToolCart().getResponsible();
        }
        if (item instanceof ToolCart) {
            ToolCart tc = (ToolCart) item;
            return tc.getResponsible();
        }
        if (item instanceof MachineComponent) {
            MachineComponent c = (MachineComponent) item;
            if (c.getResponsible() != null)
                return c.getResponsible();
            if (c.getParent() != null)
                return c.getParent().getResponsible();
        }
        return null;
    }
}
```

L'objectif de CHAIN OF RESPONSABILITY est d'exonérer le code appelant de l'obligation de savoir quel objet peut traiter une requête. Dans cet exemple, l'appelant est un menu et la requête concerne l'identification d'un ingénieur responsable. Dans la conception actuelle, l'appelant doit connaître les éléments qui possèdent une méthode `getResponsible()`. Vous pouvez perfectionner ce code en appliquant CHAIN OF RESPONSABILITY, en donnant à tous les éléments simulés un tiers responsable. Ainsi, ce sont les objets simulés qui ont pour charge de connaître leur responsable et non plus le menu.

Exercice 12.2

Redessinez le diagramme de la Figure 12.2 en déplaçant la méthode `getResponsible()` vers `VisualizationItem` et en ajoutant ce comportement à `Tool`.

Le code de menu devient plus simple maintenant qu'il peut demander à chaque élément pouvant être sélectionné son ingénieur responsable :

```
package com.oozinoz.machine;

public class AmbitiousMenu2 {
    public Engineer getResponsible(VisualizationItem item) {
        return item.getResponsible();
    }
}
```

L'implémentation de la méthode `getResponsible()` pour chaque élément est également simple.

Exercice 12.3

Ecrivez le code de la méthode `getResponsible()` pour :

- A. MachineComponent
- B. Tool
- C. ToolCart

Ancrage d'une chaîne de responsabilités

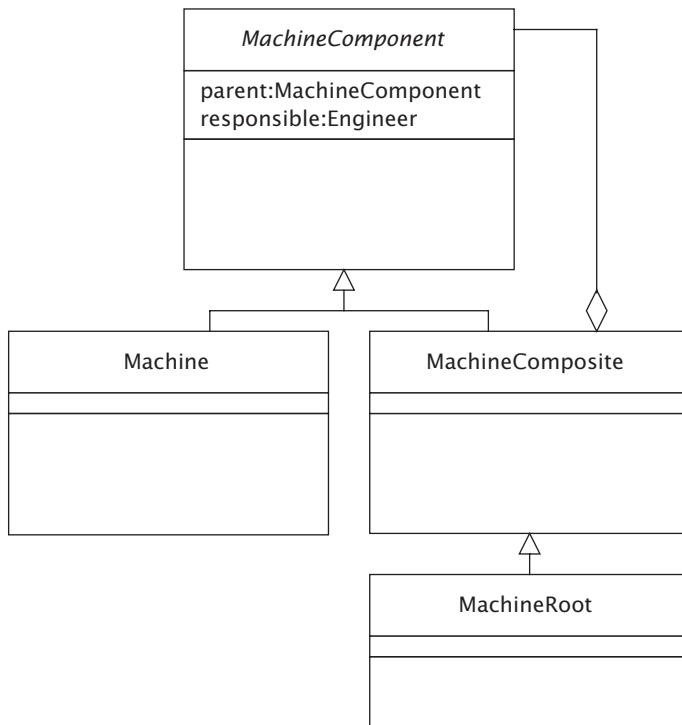
Lorsque vous écrivez la méthode `getResponsible()` pour `MachineComponent`, vous devez considérer le fait que le parent d'un objet `MachineComponent` puisse être `null`. Une autre solution est d'être un peu plus strict dans votre modèle objet et d'exiger que les objets `MachineComponent` aient un parent non `null`. Pour cela, vous pouvez ajouter un argument `parent` au constructeur de `MachineComponent`. Vous pouvez même émettre une exception lorsque l'objet fourni est `null`, tant que vous savez où cette exception est interceptée. Considérez aussi le fait qu'un objet formera la racine (*root*) — un objet particulier qui n'aura pas de parent. Une approche raisonnable est de créer une classe `MachineRoot` en tant que sous-classe de `MachineComposite` (pas de `MachineComponent`). Vous pouvez alors garantir qu'un objet `MachineComponent` aura toujours un ingénieur responsable si :

- Le constructeur (ou les constructeurs) de `MachineRoot` requiert un objet `Engineer`.
- Le constructeur (ou les constructeurs) de `MachineComponent` requiert un objet parent qui soit lui-même un `MachineComponent`.

- Seul MachineRoot utilise null comme valeur pour son parent.

Figure 12.3

Comment les constructeurs peuvent-ils garantir que chaque objet Machine-Component aura un ingénieur responsable ?



Exercice 12.4

Complétez les constructeurs de la Figure 12.3 pour supporter une conception garantissant que chaque objet MachineComponent aura un ingénieur responsable.

En ancrant une chaîne de responsabilités, vous renforcez le modèle objet et simplifiez le code. Vous pouvez maintenant implémenter la méthode `getResponsible()` de **MachineComponent** comme suit :

```
public Engineer getResponsible() {  
    if (responsible != null)  
        return responsible;  
    return parent.getResponsible();  
}
```

CHAIN OF RESPONSABILITY sans COMPOSITE

Le pattern CHAIN OF RESPONSABILITY requiert une stratégie pour ordonner la recherche d'un objet pouvant traiter une requête. Généralement, l'ordre à suivre dépendra d'un aspect sous-jacent du domaine modélisé. Cela se produit souvent lorsqu'il y a une sorte de composition, comme dans la hiérarchie de composants matériels d'Oozinoz. Ce pattern peut toutefois s'appliquer à d'autres modèles que les modèles composites.

Exercice 12.5

Donnez un exemple dans lequel le pattern CHAIN OF RESPONSABILITY peut intervenir alors que les objets chaînés ne forment pas un composite.

Résumé

Lorsque vous appliquez le pattern CHAIN OF RESPONSABILITY, vous dispensez un client de devoir savoir quel objet d'un ensemble supporte un certain comportement. En permettant à l'action de recherche de responsabilité de se produire le long de la chaîne d'objets, vous dissociez le client de tout objet spécifique de la chaîne.

Ce pattern intervient occasionnellement lorsqu'une chaîne d'objets arbitraire peut appliquer une série de stratégies diverses pour répondre à un certain problème, tel que l'analyse d'une entrée utilisateur. Plus fréquemment, il intervient dans le cas d'agrégats, où une hiérarchie d'isolement fournit un ordre naturel pour une chaîne d'objets. Ce pattern résulte en un code plus simple au niveau à la fois de la hiérarchie et du client

FLYWEIGHT

Le pattern **FLYWEIGHT** permet le partage d'un objet entre plusieurs clients, créant une responsabilité pour l'objet partagé dont les objets ordinaires n'ont normalement pas à se soucier. La plupart du temps, un seul client à la fois détient une référence vers un objet. Lorsque l'état de l'objet change, c'est parce que le client l'a modifié et l'objet n'a pas la responsabilité d'en informer les autres clients. Il est cependant parfois utile de pouvoir partager l'accès à un objet.

Une raison de vouloir cela apparaît lorsque vous devez gérer des milliers ou des dizaines de milliers de petits objets, tels que les caractères d'une version en ligne d'un livre. Dans un tel cas, ce sera pour améliorer les performances afin de pouvoir partager efficacement des objets d'une grande granularité entre de nombreux clients. Un livre n'a besoin que d'un objet A, bien qu'il nécessite un moyen de modéliser les endroits où différents A apparaissent.

L'objectif du pattern FLYWEIGHT est d'utiliser le partage pour supporter efficacement un grand nombre d'objets à forte granularité.

Immuabilité

Le pattern **FLYWEIGHT** laisse plusieurs clients se partager un grand nombre de petits objets : les *flyweights* (poids mouche). Pour que cela fonctionne, vous devez considérer que lorsqu'un client change l'état d'un objet, cet état est modifié pour chaque client ayant accès à l'objet. La façon la plus simple et la plus courante d'éviter qu'ils se perturbent mutuellement est de les empêcher d'introduire des changements d'état dans l'objet partagé. Un moyen d'y parvenir est de créer un objet qui soit **immutable** pour que, une fois créé, il ne puisse être changé. Les objets immuables

les plus fréquemment rencontrés dans Java sont des instances de la classe `String`. Une fois que vous avez créé une chaîne, ni vous ni aucun client pouvant y accéder ne pourra changer ses caractères.

Exercice 13.1

Donnez une justification du choix des créateurs de Java d'avoir rendu les objets `String` immuables, ou argumentez contre cette décision si vous la jugez déraisonnable.

- *Les solutions des exercices de ce chapitre sont données dans l'Annexe B.*

Lorsque vous avez un grand nombre d'objets similaires, vous pouvez vouloir en partager l'accès, mais ils ne sont pas nécessairement immuables. Dans ce cas, une étape préalable à l'application de FLYWEIGHT est d'extraire la partie immuable d'un objet pour qu'elle puisse être partagée.

Extraction de la partie immuable d'un flyweight

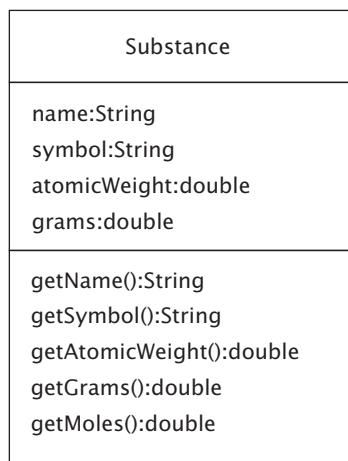
Chez Oozinoz, les substances chimiques sont aussi répandues que des caractères dans un document. Les services achat, ingénierie, fabrication et sécurité sont tous impliqués dans la gestion de la circulation de milliers de substances chimiques dans l'usine. Les préparations chimiques sont souvent modélisées au moyen d'instances de la classe `Substance` illustrée Figure 13.1.

La classe `Substance` possède de meilleures méthodes pour ses attributs ainsi qu'une méthode `getMoles()` qui retourne le nombre de **moles** — un compte de molécules — dans une substance. Un objet `Substance` représente une certaine quantité d'une certaine molécule. Oozinoz utilise une classe `Mixture` pour modéliser des combinaisons de substances. Par exemple, la Figure 13.2 présente un diagramme d'une préparation de poudre noire.

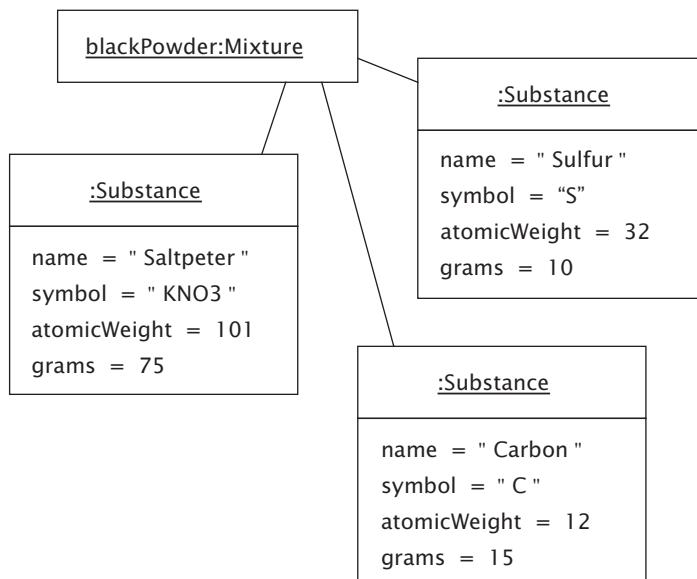
Supposez que, étant donné la prolifération de substances chimiques chez Oozinoz, vous décidiez d'appliquer le pattern FLYWEIGHT pour réduire le nombre d'objets `Substance` dans les applications. Pour traiter les objets `Substance` en tant que flyweights, une première étape est de séparer les parties immuables des parties variables. Supposez que vous décidiez de restructurer la classe `Substance` en extrayant sa partie immuable pour la placer dans une classe `Chemical`.

Figure 13.1

Un objet Substance modélise une préparation chimique.

**Figure 13.2**

Une préparation de poudre noire contient du salpêtre, du soufre et du charbon.



Exercice 13.2

Complétez le diagramme de classes de la Figure 13.3 pour présenter une classe Substance2 restructurée et une nouvelle classe Chemical immuable.

Figure 13.3

Complétez ce diagramme pour extraire les caractéristiques immuables de Substance2 et les placer dans la classe Chemical.

Substance2	Chemical
?? ...	?? ...
?? ...	?? ...

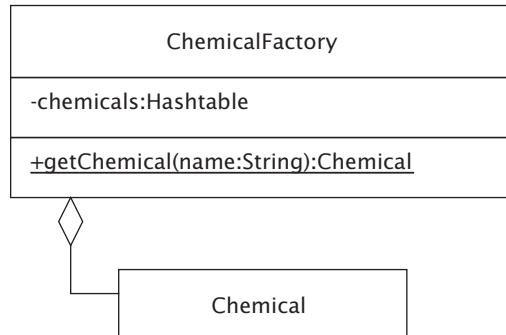
Partage des objets flyweight

Extraire la partie immuable d'un objet n'est qu'une partie du travail dans l'application du pattern FLYWEIGHT. Vous devez encore créer une classe factory flyweight qui instancie les flyweights, et faire en sorte que les clients se les partagent. Vous devez aussi vous assurer que les clients utiliseront votre factory au lieu de construire eux-mêmes des instances de la classe flyweight.

Pour créer des flyweights, vous avez besoin d'une factory, peut-être une classe ChemicalFactory avec une méthode statique qui retourne une substance chimique d'après un nom donné. Vous pourriez stocker les substances dans une table de hachage, créant des substances connues lors de l'initialisation de la factory. La Figure 13.4 illustre un exemple de conception pour ChemicalFactory.

Figure 13.4

La classe ChemicalFactory est une factory flyweight qui retourne des objets Chemical.



Le code de ChemicalFactory peut utiliser un initialisateur statique pour stocker les objets Chemical dans une table Hashtable :

```

package com.oozinoz.chemical;
import java.util.*;
  
```

```
public class ChemicalFactory {  
    private static Map chemicals = new HashMap();  
  
    static {  
        chemicals.put(  
            "carbon", new Chemical("Carbon", "C", 12));  
        chemicals.put(  
            "sulfur", new Chemical("Sulfur", "S", 32));  
        chemicals.put(  
            "saltpeter", new Chemical("Saltpeter", "KN03", 101));  
        //...  
    }  
  
    public static Chemical getChemical(String name) {  
        return (Chemical) chemicals.get(name.toLowerCase());  
    }  
}
```

Après avoir créé une factory pour les substances chimiques, vous devez maintenant prendre des mesures pour vous assurer que d'autres développeurs l'utiliseront et n'instancieront pas eux-mêmes la classe `Chemical`. Une approche simple est de s'appuyer sur l'accessibilité de la classe `Chemical`.

Exercice 13.3

Comment pouvez-vous utiliser l'accessibilité de la classe `Chemical` pour décourager d'autres développeurs de l'instancier ?

Les modificateurs d'accès ne fournissent pas le contrôle total sur l'instanciation dont vous auriez besoin. Vous pourriez vous assurer que `ChemicalFactory` soit la seule classe à pouvoir créer de nouvelles instances `Chemical`. Pour atteindre ce niveau de contrôle, vous pouvez appliquer une classe *interne* en définissant la classe `Chemical` dans `ChemicalFactory` (voir le package `com.oozinoz.chemical2`).

Pour accéder à un type imbriqué, les clients doivent spécifier le type "contenant", avec des expressions telles que les suivantes :

```
ChemicalFactory.Chemical c =  
    ChemicalFactory.getChemical("saltpeter");
```

Vous pouvez simplifier l'emploi d'une classe imbriquée en faisant de Chemical une interface et en nommant la classe ChemicalImpl. L'interface Chemical peut spécifier trois méthodes accesseurs, comme suit :

```
package com.oozinoz.chemical2;
public interface Chemical {
    String getName();
    String getSymbol();
    double getAtomicWeight();
}
```

Les clients ne référenceront jamais directement la classe interne. Vous pouvez donc la définir privée pour avoir la garantie que seul ChemicalFactory2 y aura accès.

Exercice 13.4

Complétez le code suivant pour ChemicalFactory2.java.

```
package com.oozinoz.chemical2;
import java.util.*;

public class ChemicalFactory2 {
    private static Map chemicals = new HashMap();

    /* Exercice ! */ implements Chemical {
        private String name;
        private String symbol;
        private double atomicWeight;

        ChemicalImpl(
            String name,
            String symbol,
            double atomicWeight) {
            this.name = name;
            this.symbol = symbol;
            this.atomicWeight = atomicWeight;
        }

        public String getName() {
            return name;
        }

        public String getSymbol() {
            return symbol;
        }
    }
}
```

```
public double getAtomicWeight() {
    return atomicWeight;
}

public String toString() {
    return name + "(" + symbol + ")[ " +
        atomicWeight + " ]";
}
}

/* Exercice ! */
chemicals.put("carbon",
    factory.new ChemicalImpl("Carbon", "C", 12));
chemicals.put("sulfur",
    factory.new ChemicalImpl("Sulfur", "S", 32));
chemicals.put("saltpeter",
    factory.new ChemicalImpl(
        "Saltpeter", "KN03", 101));
//...
}

public static Chemical getChemical(String name) {
    return /* Exercice ! */
}
}
```

Résumé

Le pattern FLYWEIGHT vous permet de partager l'accès à des objets qui peuvent se présenter en grande quantité, tels que des caractères ou des substances chimiques. Les objets flyweight doivent être immuables, une propriété que vous pouvez établir en extrayant la partie immuable de la classe que vous voulez partager. Pour garantir le partage des objets flyweight, vous pouvez fournir une classe factory à partir de laquelle les clients pourront obtenir des flyweights, puis forcer l'emploi de cette factory. Les modificateurs d'accès vous donnent un certain contrôle sur l'accès à votre code par les autres développeurs, mais vous bénéficiez d'un meilleur contrôle au moyen de classes internes en garantissant qu'une classe ne pourra être accessible que par la classe qui la contient. En vous assurant que les clients utiliseront comme il se doit votre factory flyweight, vous pouvez fournir un accès partagé sécurisé à de nombreux objets.

III

Patterns de construction

14

Introduction à la construction

Lorsque vous créez une classe Java, vous prévoyez normalement une fonctionnalité pour la création des objets en fournissant un **constructeur**. Un constructeur est cependant utile uniquement si les clients savent quelle classe instancier et disposent des paramètres que le constructeur attend. Plusieurs patterns de conception peuvent intervenir dans les situations où ces conditions, ou d'autres circonstances de construction ordinaire, ne valent pas. Avant d'examiner ces types de conception utiles où la construction ordinaire ne suffit pas, il peut être utile de revoir ce qu'est une construction classique en Java.

Quelques défis de construction

Les constructeurs sont des méthodes spéciales. Par bien des aspects, dont les modificateurs d'accès, la surcharge ou les listes de paramètres, les constructeurs s'apparentent à des méthodes ordinaires. D'un autre côté, leur emploi et leur comportement sont régis par un nombre significatif de règles syntaxiques et sémantiques.

Exercice 14.1

Citez quatre règles gouvernant l'usage et le comportement des constructeurs dans le langage Java.

- *Les solutions des exercices de ce chapitre sont données dans l'Annexe B.*

Dans certains cas, Java fournit des constructeurs avec un comportement par défaut. Tout d'abord, si une classe ne possède pas de constructeur déclaré, Java en fournit un par défaut, lequel équivaut à un constructeur public, n'attendant aucun argument et ne comportant aucune instruction dans son corps.

Un second comportement par défaut du langage se produit lorsque la déclaration du constructeur d'une classe n'utilise pas une variation de `this()` ou de `super()` pour invoquer de façon explicite un autre constructeur. Java insère alors `super()` sans argument. Cela peut provoquer des résultats surprenants, comme avec la compilation du code suivant :

```
package app.construction;
public class Fuse {
    private String name;
    // public Fuse(String name) { this.name = name; }
}
```

et :

```
package app.construction;
public class QuickFuse extends Fuse { }
```

Ce code compile correctement tant que vous ne retirez pas les marques de commentaire `//`.

Exercice 14.2

Expliquez l'erreur qui se produira si vous retirez les marques de commentaire, permettant ainsi à la super-classe `Fuse` d'accepter un nom dans son constructeur.

La façon la plus courante d'instancier des objets est d'invoquer l'opérateur `new`, mais vous pouvez aussi utiliser la **réflexion**. La réflexion donne la possibilité de travailler avec des types et des membres de type en tant qu'objets. Même si vous n'utilisez pas fréquemment la réflexion, il n'est pas trop difficile de suivre la logique d'un programme s'appuyant sur cette technique, comme dans l'exemple suivant :

```
package app.construction;
import java.awt.Point;
import java.lang.reflect.Constructor;
```

```
public class ShowReflection {  
    public static void main(String args[]) {  
        Constructor[] cc = Point.class.getConstructors();  
        Constructor cons = null;  
        for (int i = 0; i < cc.length; i++)  
            if (cc[i].getParameterTypes().length == 2)  
                cons = cc[i];
```

Exercice 14.3

Qu'est-ce que le programme ShowReflection produit en sortie ?

La réflexion vous permet d'atteindre des résultats qui sont autrement difficiles ou impossibles à atteindre.

Résumé

D'ordinaire, vous fournissez des classes avec des constructeurs pour en permettre l'instanciation. Ceux-ci peuvent former une suite collaborative et chaque constructeur doit au final invoquer le constructeur de la super-classe. La méthode classique d'appel d'un constructeur est l'emploi de l'opérateur new mais vous pouvez aussi recourir à la réflexion pour instancier et utiliser des objets.

Au-delà de la construction ordinaire

Le mécanisme de constructeur dans Java offre de nombreuses options de conception de classe. Toutefois, un constructeur d'une classe n'est efficace que si l'utilisateur sait quelle classe instancier et connaît les champs requis pour l'instanciation. Par exemple, le choix des composants d'une GUI à créer peut dépendre du matériel sur lequel le programme doit s'exécuter. Un équipement portable n'aura pas la même surface d'affichage qu'un ordinateur. Il peut aussi arriver qu'un développeur sache quelle classe instancier mais ne possède pas toutes les valeurs initiales, ou qu'il les ait dans le mauvais format. Par exemple, le développeur peut avoir besoin de créer un objet à partir d'une version dormante ou textuelle d'un objet. Dans une telle situation, l'emploi ordinaire de constructeurs Java ne suffit pas et vous devez recourir à un pattern de conception.

Le tableau suivant décrit l'objectif de patterns qui facilitent la construction.

<i>Si vous envisagez de</i>	<i>Appliquez le pattern</i>
• Collecter progressivement des informations sur un objet avant de demander sa construction	BUILDER
• Différer la décision du choix de la classe à instancier	FACTORY METHOD
• Construire une famille d'objets qui partagent certains aspects	ABSTRACT FACTORY
• Spécifier un objet à créer en donnant un exemple	PROTOTYPE
• Reconstruire un objet à partir d'une version dormante ne contenant que l'état interne de l'objet	MEMENTO

L'objectif de chaque pattern de conception est de permettre la résolution d'un problème dans un certain contexte. Les patterns de construction permettent à un client de construire un nouvel objet par l'intermédiaire de moyens autres que l'appel d'un constructeur de classe. Par exemple, lorsque vous obtenez progressivement les valeurs initiales d'un objet, vous pouvez envisager d'appliquer le pattern BUILDER.

BUILDER

Vous ne disposez pas toujours de toutes les informations nécessaires pour créer un objet lorsque vient le moment de le construire. Il est particulièrement pratique de permettre la construction progressive d'un objet, au rythme de l'obtention des paramètres pour le constructeur, comme cela se produit avec l'emploi d'un analyseur syntaxique ou avec une interface utilisateur. Cela peut aussi être utile lorsque vous souhaitez simplement réduire la taille d'une classe dont la construction est relativement compliquée sans que cette complexité ait réellement de rapport avec le but principal de la classe.

L'objectif du pattern BUILDER est de déplacer la logique de construction d'un objet en dehors de la classe à instancier.

Un objet constructeur ordinaire

Une situation banale dans laquelle vous pouvez tirer parti du pattern BUILDER est celle où les données qui définissent l'objet voulu sont incorporées dans une chaîne de texte. A mesure que votre code examine, ou analyse, les données, vous devez les stocker telles que vous les trouvez. Que votre analyseur s'appuie sur XML ou soit une création personnelle, il est possible que vous ne disposiez initialement pas de suffisamment de données pour construire l'objet voulu. La solution fournie par BUILDER est d'enregistrer les données extraites du texte dans un objet intermédiaire jusqu'à ce que le programme soit prêt à lui demander de construire l'objet à partir de ces données.

Supposez qu'en plus de fabriquer des fusées, Oozinoz organise parfois des feux d'artifice. Les agences de voyages envoient des requêtes de réservation dans le format suivant :

```
Date, November 5, Headcount, 250, City, Springfield,  
DollarsPerHead, 9,95, HasSite, False
```

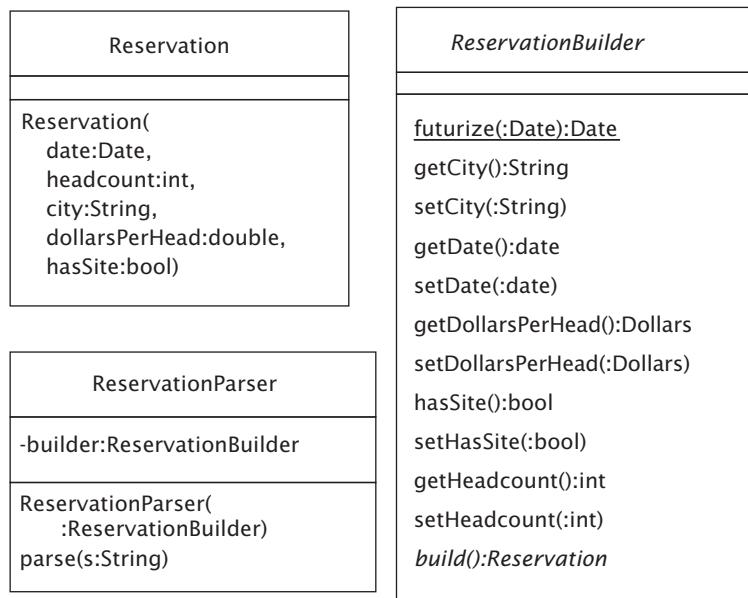
Comme vous l'avez sans doute remarqué, ce protocole remonte à une époque antérieure à XML (*Extensible Markup Language*), mais il s'est montré suffisant jusqu'à présent.

La requête signale quand un client potentiel souhaite organiser un feu d'artifice et dans quelle ville cela doit se passer. Elle spécifie aussi le nombre de personnes (Headcount) minimal garanti par le client et le prix par tête (DollarsPerHead) que le client accepte de payer. Le client, dans cet exemple, souhaite organiser un show pour 250 invités et est prêt à payer \$9,95 par personne, soit un total de \$2 487,50. L'agence de voyages indique aussi que le client n'a pas de site à l'esprit (False) pour le déroulement du show.

La tâche à réaliser consiste à analyser le texte de la requête et à créer un objet `Reservation` représentant celle-ci. Nous pourrions accomplir cela en créant un objet `Reservation` vide et en définissant ses paramètres à mesure que notre analyseur (*parser*) les rencontre. Le problème est qu'un objet `Reservation` pourrait ne pas représenter une requête valide. Par exemple, nous pourrions terminer l'analyse du texte et réaliser qu'il manque une date.

Pour nous assurer qu'un objet `Reservation` représente toujours une requête valide, nous pouvons utiliser une classe `ReservationBuilder`. L'objet `ReservationBuilder` peut stocker les attributs d'une requête de réservation à mesure que l'analyseur les trouve, puis créer un objet `Reservation` en vérifiant sa validité. La Figure 15.1 illustre les classes dont nous avons besoin pour cette conception.

La classe `ReservationBuilder` est abstraite ainsi que sa méthode `build()`. Nous créerons des sous-classes `ReservationBuilder` concrètes qui varieront au niveau de l'insistance avec laquelle elles tentent de créer un objet `Reservation` lorsque les données sont incomplètes. Le constructeur de la classe `ReservationParser` attend un builder — *NDT : nous utiliserons ce terme pour différencier l'objet de stockage du constructeur traditionnel* — auquel passer des informations.

**Figure 15.1**

Une classe builder libère une classe spécifique de la logique de construction et peut accepter progressivement des paramètres d'initialisation à mesure qu'un analyseur syntaxique les découvre.

La méthode `parse()` extrait des informations d'une chaîne de réservation et les transmet au builder, comme dans l'extrait suivant :

```

public void parse(String s) throws ParseException {
    String[] tokens = s.split(",");
    for (int i = 0; i < tokens.length; i += 2) {
        String type = tokens[i];
        String val = tokens[i + 1];

        if ("date".compareToIgnoreCase(type) == 0) {
            Calendar now = Calendar.getInstance();
            DateFormat formatter = DateFormat.getDateInstance();
            Date d = formatter.parse(
                val + ", " + now.get(Calendar.YEAR));
            builder.setDate(ReservationBuilder.futurize(d));
        } else if ("headcount".compareToIgnoreCase(type) == 0)
            builder.setHeadcount(Integer.parseInt(val));
    }
}

```

```

        else if ("City".compareToIgnoreCase(type) == 0)
            builder.setCity(val.trim());
        else if ("DollarsPerHead".compareToIgnoreCase(type)==0)
            builder.setDollarsPerHead(
                new Dollars(Double.parseDouble(val)));
        else if ("HasSite".compareToIgnoreCase(type) == 0)
            builder.setHasSite(val.equalsIgnoreCase("true"));
    }
}

```

Le code de `parse()` utilise une méthode `String.split()` pour diviser, ou découper, la chaîne fournie en entrée. Le code attend une réservation sous forme d'une liste de types d'information et de valeurs séparés par une virgule. La méthode `String.compareToIgnoreCase()` permet à la comparaison de ne pas tenir compte de la casse. Lorsque l'analyseur rencontre le mot "date", il examine la valeur qui suit et la place dans le futur. La méthode `futurize()` avance l'année de la date jusqu'à ce que cette dernière soit située dans le futur. A mesure que vous progressez dans l'examen du code, vous remarquerez plusieurs endroits où l'analyseur pourrait s'égarer, à commencer par le découpage initial de la chaîne de réservation.

Exercice 15.1

L'objet d'expression régulière utilisé par les appels de `split()` divise une liste de valeurs séparées par des virgules en chaînes individuelles. Suggérez une amélioration de cette expression régulière, ou de l'ensemble de l'approche, qui permettra à l'analyseur de mieux reconnaître les informations de réservation.

- *Les solutions des exercices de ce chapitre sont données dans l'Annexe B.*

Construction avec des contraintes

Vous devez vous assurer que les objets `Reservation` invalides ne soient jamais instanciés. Plus spécifiquement, supposez que toute réservation doit avoir une valeur non nulle pour la date et la ville. Supposez aussi qu'une règle métier stipule qu'Oozinoz ne réalisera pas le show pour moins de 25 personnes ou moins de \$495,95. Ces limites pourraient être enregistrées dans une base de données, mais pour l'instant nous les représenterons sous forme de constantes dans le code Java, comme dans l'exemple suivant :

```

public abstract class ReservationBuilder {
    public static final int MINHEAD = 25;

```

```

public static final Dollars MINTOTAL = new Dollars(495.95);
// ...
}

```

Pour éviter la création d'une instance de `Reservation` lorsqu'une requête est invalide, vous pourriez placer les contrôles de logique métier et les générations d'exceptions dans le constructeur pour `Reservation`. Cette logique est toutefois relativement indépendante de la fonction normale d'un objet `Reservation` une fois celui-ci créé. L'introduction d'un builder permettra de simplifier la classe `Reservation` en ne laissant que des méthodes dédiées à d'autres fonctions que la construction. L'emploi d'un builder donne aussi la possibilité de valider les paramètres d'un objet `Reservation` en proposant des réactions différentes en cas de paramètres invalides. Finalement, déplacé au niveau d'une sous-classe `ReservationBuilder`, le travail de construction peut se dérouler progressivement à mesure que l'analyseur découvre les valeurs des attributs de réservation. La Figure 15.2 illustre des sous-classes `ReservationBuilder` concrètes qui diffèrent dans leur façon de tolérer des paramètres invalides.

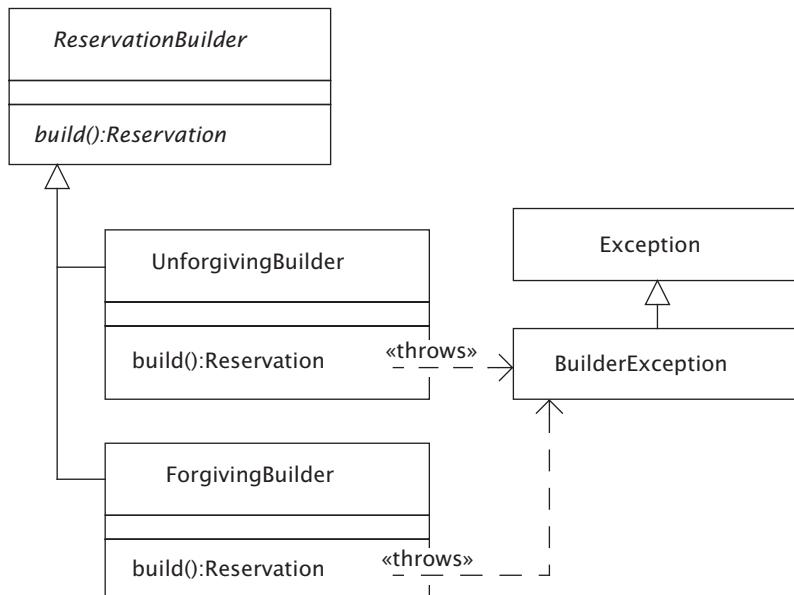


Figure 15.2

Les objets builders peuvent différer dans leur niveau de sensibilité et de génération d'exceptions en cas de chaîne de réservation incomplète.

Le diagramme de la Figure 15.2 met en valeur un avantage d'appliquer le pattern BUILDER. En séparant la logique de construction de la classe `Reservation`, nous pouvons traiter la construction comme une tâche distincte et même créer une hiérarchie d'approches distincte. Les différences de comportement lors de la construction ont peu de rapport avec la logique de réservation. Par exemple, les builders dans la Figure 15.2 diffèrent dans leur niveau de sensibilité pour ce qui est de la génération d'une exception `BuilderException`. Un code utilisant un builder ressemblera à l'extrait suivant :

```
package app.builder;
import com.oozinoz.reservation.*;

public class ShowUnforgiving {
    public static void main(String[] args) {
        String sample =
            "Date, November 5, Headcount, 250, "
            + "City, Springfield, DollarsPerHead, 9.95, "
            + "HasSite, False";
        ReservationBuilder builder = new UnforgivingBuilder();
        try {
            new ReservationParser(builder).parse(sample);
            Reservation res = builder.build();
            System.out.println("Builder non tolérant : " + res);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

L'exécution de ce programme affiche un objet `Reservation` :

```
Date: Nov 5, 2001, Headcount: 250, City: Springfield,
Dollars/Head: 9.95, Has Site: false
```

A partir d'une chaîne de requête de réservation, le code instancie un builder et un analyseur, et demande à celui-ci d'analyser la chaîne. A mesure qu'il lit la chaîne, l'analyseur transmet les attributs de réservation au builder en utilisant ses méthodes `set`.

Après l'analyse, le code demande au builder de construire une réservation valide. Cet exemple affiche simplement le texte d'un message d'exception au lieu d'entreprendre une action plus conséquente comme ce serait le cas pour une réelle application.

Exercice 15.2

La méthode `build()` de la classe `UnforgivingBuilder` génère une exception `BuilderException` si la valeur de la date ou de la ville est `null`, si le nombre de personnes est trop bas, ou si le coût total de la réservation proposée est trop faible. Ecrivez le code de la méthode `build()` en fonction de ces spécifications.

Un builder tolérant

La classe `UnforgivingBuilder` rejette toute requête comportant la moindre erreur. Une meilleure règle de gestion serait d'apporter des changements raisonnables aux requêtes auxquelles il manque certains détails concernant la réservation.

Supposez qu'un analyste d'Oozinoz vous demande de définir le nombre de personnes à un minimum si cette valeur d'attribut est omise. De même, si le prix accepté par tête est manquant, le builder pourrait définir cet attribut pour que le coût total soit supérieur au minimum requis. Ces exigences sont simples, mais la conception nécessite quelque réflexion. Par exemple, que devra faire le builder si une chaîne de réservation fournit un coût par tête sans indiquer le nombre de personnes ?

Exercice 15.3

Rédigez une spécification pour `ForgivingBuilder.build()` en prévoyant ce que le builder devrait faire en cas d'omission du nombre de personnes ou du prix par tête.

Exercice 15.4

Après avoir revu votre approche, écrivez le code de la méthode `build()` pour la classe `ForgivingBuilder`.

Les classes `ForgivingBuilder` et `UnforgivingBuilder` garantissent que les objets `Reservation` seront toujours valides. Votre conception apporte aussi de la souplesse quant à l'action à entreprendre en cas de problème dans la construction d'une réservation.

Résumé

Le pattern BUILDER sépare la construction d'un objet complexe de sa représentation. Il s'ensuit une simplification du processus de construction. Il permet à une classe de se concentrer sur la construction correcte d'un objet en permettant à la classe principale de se concentrer sur le fonctionnement d'une instance valide. Cela est particulièrement utile lorsque vous voulez garantir la validité d'un objet avant de l'instancier et ne souhaitez pas que la logique associée apparaisse dans le constructeur de la classe. Un objet builder rend aussi possible une construction progressive, ce qui se produit souvent lorsque vous créez un objet à partir de l'analyse d'un texte.

16

FACTORY METHOD

Lorsque vous développez une classe, vous fournissez généralement des constructeurs pour permettre aux clients de l'instancier. Cependant, un client qui a besoin d'un objet ne sait pas, ou ne devrait pas savoir, quelle classe instancier parmi plusieurs choix possibles.

L'objectif du pattern FACTORY METHOD est de laisser un autre développeur définir l'interface permettant de créer un objet, tout en gardant un contrôle sur le choix de la classe à instancier.

Un exemple classique : des itérateurs

Le pattern ITERATOR (itérateur) offre un moyen d'accéder de manière séquentielle aux éléments d'une collection (voir le Chapitre 28), mais FACTORY METHOD sous-tend souvent la création des itérateurs. La version 1.2 du JDK a introduit une interface Collection qui inclut une méthode `iterator()` ; toutes les collections l'implémentent. Cette opération évite que l'appelant ait à savoir quelle classe instancier.

Une méthode `iterator()` crée un objet qui retourne une séquence formée des éléments d'une collection. Par exemple, le code suivant crée et affiche le contenu d'une liste :

```
package app.factoryMethod;
import java.util.*;

public class ShowIterator {
    public static void main(String[] args) {
        List list = Arrays.asList(
            new String[] {
                "fountain", "rocket", "sparkler"});
    }
}
```

```
Iterator iter = list.iterator();
while (iter.hasNext())
    System.out.println(iter.next());
}
```

Exercice 16.1

Quelle est la classe réelle de l'objet Iterator dans ce code ?

- *Les solutions des exercices de ce chapitre sont données dans l'Annexe B.*

Le pattern FACTORY METHOD décharge le client du souci de savoir quelle classe instancier.

Identification de FACTORY METHOD

Vous pourriez penser que n'importe quelle méthode créant et retournant un nouvel objet est forcément une méthode factory. Cependant, dans la programmation orientée objet, les méthodes qui retournent de nouveaux objets sont chose courante, et elles ne sont pas toutes des instances de FACTORY METHOD.

Exercice 16.2

Nommez deux méthodes fréquemment utilisées des bibliothèques de classes Java qui retournent un nouvel objet.

Le fait qu'une méthode crée un nouvel objet ne signifie pas nécessairement qu'il s'agit d'un exemple de FACTORY METHOD. Une méthode factory est une opération qui non seulement produit un nouvel objet mais évite au client de savoir quelle classe instancier. Dans une conception FACTORY METHOD, vous trouvez plusieurs classes qui implémentent la même opération retournant le même type abstrait, mais, lors de la demande de création d'un nouvel objet, la classe qui est effectivement instanciée dépend du comportement de l'objet factory recevant la requête.

Exercice 16.3

Le nom de la classe javax.swing.BorderFactory semble indiquer un exemple du pattern FACTORY METHOD. Expliquez en quoi l'objectif du pattern diffère de celui de cette classe.

Garder le contrôle sur le choix de la classe à instancier

En général, un client qui requiert un objet instancie la classe voulue en utilisant un de ses constructeurs. Il se peut aussi parfois que le client ne sache pas exactement quelle classe instancier. Cela peut se produire, par exemple, dans le cas d'itérateurs, la classe requise dépendant du type de collection que le client souhaite parcourir, mais aussi fréquemment dans du code d'application.

Supposez qu'Oozinoz soit prêt à laisser les clients acheter des feux d'artifice à crédit. Dès le début du développement du système d'autorisation de crédit, vous acceptez de prendre en charge la conception d'une classe `CreditCheckOnline` dont l'objectif sera de vérifier si un client peut disposer d'un certain montant de crédit chez Oozinoz.

En entamant le développement, vous réalisez que l'organisme de crédit sera parfois hors ligne. L'analyste du projet détermine que, dans ce cas, il faut que le représentant du centre de réception des appels puisse disposer d'une boîte de dialogue pour prendre une décision sur la base de quelques questions.

Vous créez donc une classe `CreditCheckOffline` et implémentez le processus en respectant les spécifications. Initialement, vous concevez les classes comme illustré Figure 16.1. La méthode `creditLimit()` accepte un numéro d'identification de client et retourne sa limite de crédit.

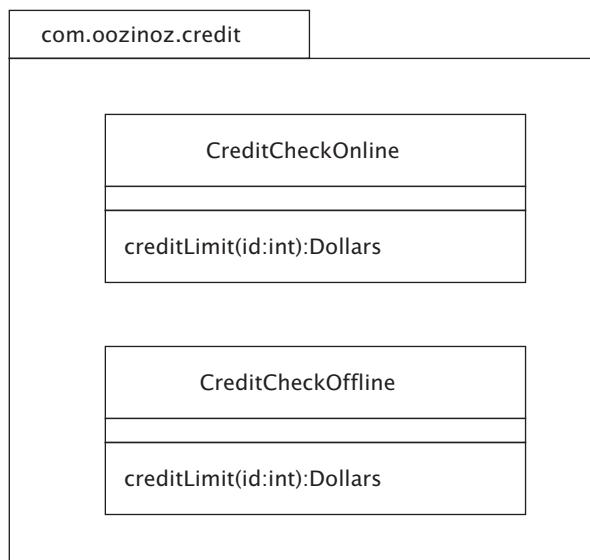
Avec les classes de la Figure 16.1, vous pouvez fournir des informations de limite de crédit, que l'organisme de crédit soit ou non en ligne. Le problème qui se présente maintenant est que l'utilisateur de vos classes doit connaître la classe à instancier, mais *vous* êtes celui qui sait si l'organisme est ou non disponible.

Dans ce scénario, vous devez vous appuyer sur l'interface pour créer un objet, mais garder le contrôle sur le choix de la classe à instancier. Une solution possible est de changer les deux classes pour implémenter une interface standard et créer une méthode factory qui retourne un objet de ce type. Spécifiquement, vous pourriez :

- faire une interface Java `CreditCheck` qui inclut la méthode `creditLimit()` ;
- changer les deux classes de contrôle de crédit afin qu'elles implémentent l'interface `CreditCheck` ;
- créer une classe `CreditCheckFactory` avec une méthode `createCreditCheck()` qui retournera un objet de type `CreditCheck`.

Figure 16.1

Une de ces classes sera instanciée pour vérifier la limite de crédit d'un client.



En implémentant `createCreditCheck()`, vous utiliserez vos informations de disponibilité de l'organisme de crédit pour décider de la classe à instancier.

Exercice 16.4

Dessinez un diagramme de classes pour cette nouvelle stratégie, qui permet de créer un objet de vérification de crédit tout en conservant la maîtrise sur le choix de la classe à instancier.

Grâce à l'implémentation de `FACTORY METHOD`, l'utilisateur de vos services pourra appeler la méthode `createCreditCheck()` et obtenir un objet de contrôle de crédit qui fonctionnera indépendamment de la disponibilité de l'agence.

Exercice 16.5

Supposez que la classe `CreditCheckFactory` comprenne une méthode `isAgencyUp()` indiquant si l'agence est disponible et écrivez le code pour `createCreditCheck()`.

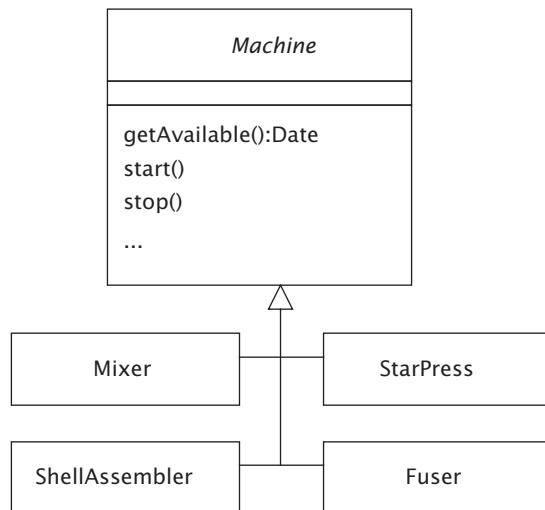
Application de FACTORY METHOD dans une hiérarchie parallèle

Le pattern FACTORY METHOD apparaît souvent lorsque vous utilisez une hiérarchie parallèle pour modéliser un domaine de problèmes. Une **hiérarchie parallèle** est une paire de hiérarchies de classes dans laquelle chaque classe d'une hiérarchie possède une classe correspondante dans l'autre hiérarchie. Une telle conception intervient généralement lorsque vous décidez de déplacer un sous-ensemble d'opérations hors d'une hiérarchie déjà existante.

Considérez la construction de bombes aériennes comme illustré au Chapitre 5. Pour les fabriquer, Oozinoz utilise des machines organisées selon le modèle du diagramme présenté à la Figure 16.2.

Figure 16.2

La hiérarchie Machine intègre une logique de contrôle des machines physiques et de planification.



Pour concevoir une bombe, des substances sont mélangées dans un mixeur (Mixer) puis passées à une presse extrudeuse (StarPress) qui produit des granules, ou étoiles. Celles-ci sont tassées dans une coque sphérique contenant en son centre de la poudre noire et le tout est placé au-dessus d'une chasse, ou charge de propulsion, au moyen d'une assembleuse (ShellAssembler). Un dispositif d'allumage est ensuite inséré (Fuser), lequel servira à la mise à feu de la charge de propulsion et à celle de la coque centrale.

Imaginez que vous vouliez que la méthode `getAvailable()` prévoie le moment où une machine termine le traitement en cours et est disponible pour un autre travail.

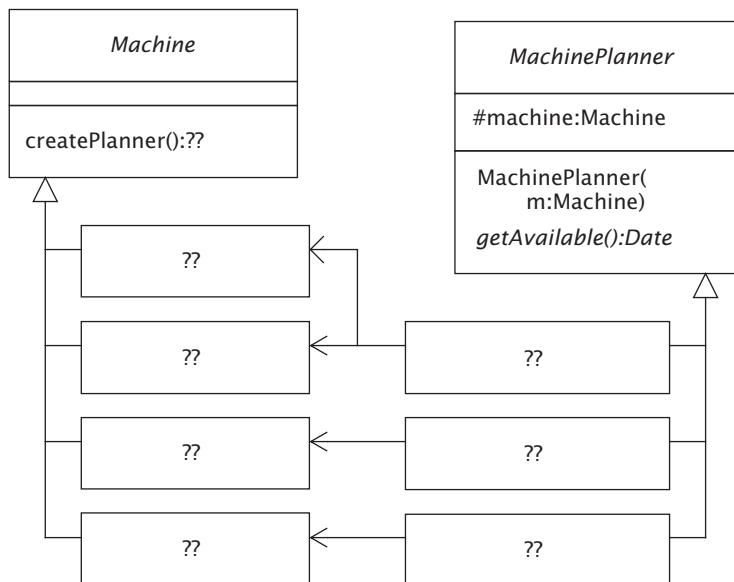
Cela peut nécessiter l'emploi de diverses méthodes privées qui ajouteront un certain volume de code à chacune de nos classes de machines. Plutôt que d'ajouter la logique de planification à la hiérarchie Machine, vous pourriez préférer utiliser une hiérarchie MachinePlanner distincte. Vous avez besoin d'une classe de planification distincte pour la plupart des types de machines, sauf pour les mixeurs et les sertisseuses de dispositifs d'allumage, qui sont toujours disponibles pour du travail supplémentaire et peuvent se suffire d'une classe BasicPlanner.

Exercice 16.6

Complétez le diagramme de la hiérarchie parallèle Machine/MachinePlanner de la Figure 16.3.

Figure 16.3

Epurez la hiérarchie Machine en déplaçant la logique de planification vers une hiérarchie parallèle.



Exercice 16.7

Ecrivez une méthode `createPlanner()` pour que la classe `Machine` retourne un objet `BasicPlanner`, et écrivez une méthode `createPlanner()` pour la classe `StarPress`.

Résumé

L'objectif du pattern FACTORY METHOD est de permettre à un fournisseur de services d'exonérer le client du besoin de savoir quelle classe instancier. Ce pattern intervient dans la bibliothèque de classes Java, notamment dans la méthode `iterator()` de l'interface `Collection`.

FACTORY METHOD se présente souvent au niveau du code du client, lorsqu'il est nécessaire de décharger les clients de la nécessité de connaître la classe à partir de laquelle créer un objet. Ce besoin d'isoler le client peut apparaître lorsque le choix de la classe à instancier dépend d'un facteur dont le client n'a pas connaissance, comme la disponibilité d'un service externe. Vous pouvez également rencontrer FACTORY METHOD lorsque vous introduisez une hiérarchie parallèle pour éviter qu'un ensemble de classes soit encombré par de nombreux aspects comportementaux. Vous pouvez ainsi relier des hiérarchies en permettant aux sous-classes d'une hiérarchie de déterminer quelle classe instancier dans la hiérarchie correspondante.

ABSTRACT FACTORY

Comme nous l'avons vu dans le chapitre précédent, il est parfois utile, lors de la création d'objets, de garder un contrôle sur le choix de la classe à instancier. Dans ce cas, vous pouvez appliquer le pattern **FACTORY METHOD** avec une méthode qui utilise un facteur externe pour déterminer la classe à instancier. Dans certaines circonstances, ce facteur peut être thématique, couvrant plusieurs classes.

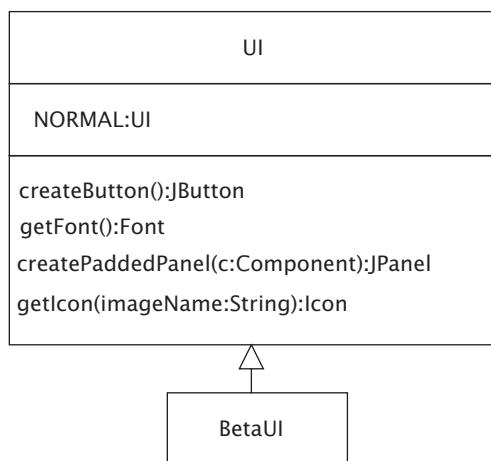
L'objectif du pattern ABSTRACT FACTORY, ou KIT, est de permettre la création de familles d'objets ayant un lien ou interdépendants.

Un exemple classique : le kit de GUI

Un kit de GUI est un exemple classique d'application du pattern **ABSTRACT FACTORY**. C'est un objet factory abstrait qui fournit les composants graphiques à un client élaborant une interface utilisateur. Il détermine l'apparence que revêtent les boutons, les champs de texte ou tout autre élément. Un kit établit un style spécifique, en définissant les couleurs d'arrière-plan, les formes, ou autres aspects d'une GUI. Vous pourriez ainsi établir un certain style pour la totalité d'un système ou, au fil du temps, introduire des changements dans une application existante, par exemple pour refléter un changement de version ou une modification des graphiques standards de la société. Ce pattern permet ainsi d'apporter de la convivialité, de contribuer à un apprentissage et une utilisation plus aisées d'une application en jouant sur son apparence. La Figure 17.1 illustre un exemple avec la classe `UI` d'Oozinoz.

Figure 17.1

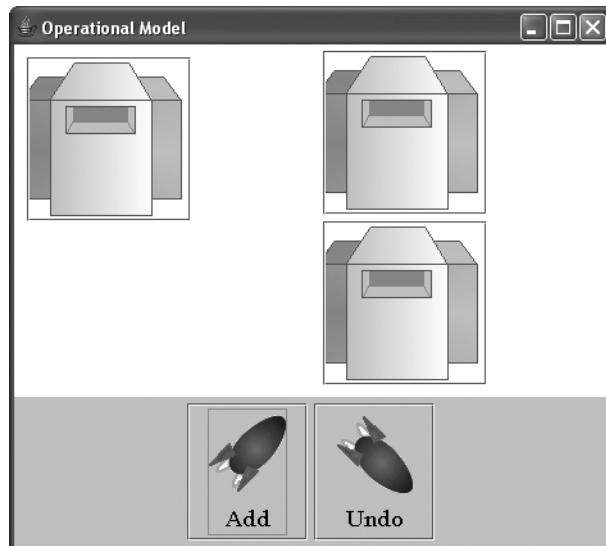
Les instances de la classe UI sont des objets factory qui créent des familles de composants de GUI.



Les sous-classes de la classe UI peuvent redéfinir n’importe quel élément de l’objet factory. Une application qui construit une GUI à partir d’une instance de la classe UI peut par la suite produire un style différent en se fondant sur une instance d’une sous-classe de UI. Par exemple, Oozinoz utilise une classe Visualization pour aider les ingénieurs à mettre en place de nouvelles lignes de fabrication. L’écran de visualisation est illustré Figure 17.2.

Figure 17.2

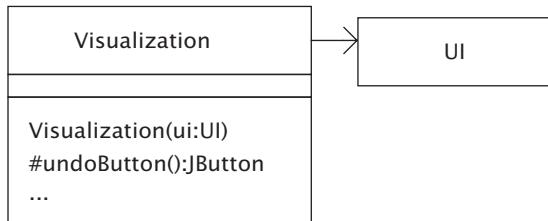
Cette application ajoute des machines dans la partie supérieure gauche de la fenêtre et laisse l’utilisateur les positionner par un glisser-déposer. Il peut annuler un ajout ou un positionnement.



L’application de visualisation de la Figure 17.2 permet à un utilisateur d’ajouter des machines et de les déplacer au moyen de la souris — le programme qui affiche cette visualisation est `ShowVisualization` dans le répertoire `app.abstractFactory`. L’application obtient ses boutons à partir d’un objet UI que la classe `Visualization` accepte dans son constructeur. La Figure 17.3 illustre la classe `Visualization`.

Figure 17.3

La classe `Visualization` construit une GUI au moyen d’un objet factory UI.



La classe `Visualization` construit sa GUI à l’aide d’un objet `UI`. Par exemple, le code de la méthode `undoButton()` se présente comme suit :

```

protected JButton undoButton() {
    if (undoButton == null) {
        undoButton = ui.createButtonCancel();
        undoButton.setText("Undo");
        undoButton.setEnabled(false);
        undoButton.addActionListener(mediator.undoAction());
    }
    return undoButton;
}
  
```

Ce code crée un bouton d’annulation et modifie son texte (pour qu’il indique “Undo”). La classe `UI` détermine la taille et la position de l’image et du texte sur le bouton. Le code générateur de bouton de la classe `UI` se présente comme suit :

```

public JButton createButton() {
    JButton button = new JButton();
    button.setSize(128, 128);
    button.setFont(getFont());
    button.setVerticalTextPosition(AbstractButton.BOTTOM);
    button.setHorizontalTextPosition(AbstractButton.CENTER);
    return button;
}

public JButton createButtonOk() {
    JButton button = createButton();
    button.setIcon(getIcon("images/rocket-large.gif"));
    button.setText("Ok!");
    return button;
}
  
```

```
public JButton createButtonCancel() {  
    JButton button = createButton();  
    button.setIcon(getIcon("images/rocket-large-down.gif"));  
    button.setText("Cancel!");  
    return button;  
}
```

Afin de générer un autre style pour l’application de visualisation des machines, nous pouvons créer une sous-classe qui redéfinit certains des éléments de la classe factory UI. Nous pourrons ensuite passer une instance de cette nouvelle classe factory au constructeur de la classe `Visualization`.

Supposez que nous ayons introduit une nouvelle version de la classe `Visualization` avec des fonctionnalités supplémentaires. Pendant sa phase de bêta-test, nous décidons de changer l’interface utilisateur. Nous aimerais en fait avoir des polices en italiques et substituer aux images de fusées des images provenant des fichiers `cherry-large.gif` et `cherry-largedown.gif`. Voici un exemple de code d’une classe `BetaUI` dérivée de `UI` :

```
public class BetaUI extends UI {  
    public BetaUI () {  
        Font oldFont = getFont();  
        font = new Font(  
            oldFont.getName(),  
            oldFont.getStyle() | Font.ITALIC,  
            oldFont.getSize());  
    }  
  
    public JButton createButtonOk() {  
        // Exercice !  
    }  
  
    public JButton createButtonCancel() {  
        // Exercice !  
    }  
}
```

Exercice 17.1

Complétez le code pour la classe `BetaUI`.

- Les solutions des exercices de ce chapitre sont données dans l’Annexe B.

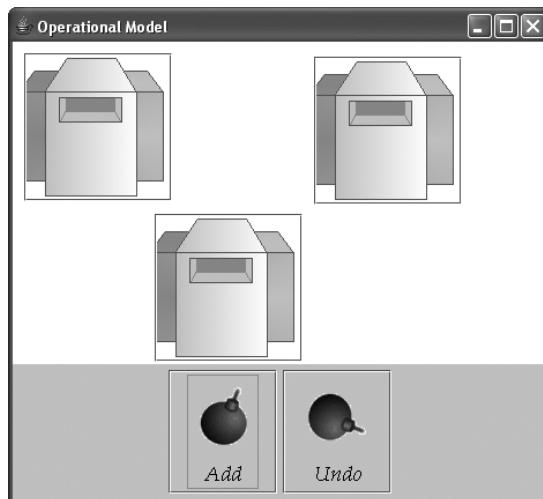
Le code suivant exécute la visualisation avec le nouveau style :

```
package app.abstractFactory;  
// ...  
public class ShowBetaVisualization {  
    public static void main(String[] args) {  
        JPanel panel = new Visualization(new BetaUI());  
        SwingFacade.launch(panel, "Operational Model");  
    }  
}
```

Ce programme exécute la visualisation avec l'apparence illustrée Figure 17.4. Les instances de UI et de BetaUI fournissent différentes familles de composants graphiques afin de proposer différents styles. Bien que ce soit une application utile du pattern ABSTRACT FACTORY, la conception est quelque peu fragile. En particulier, la classe BetaUI dépend de la possibilité de redéfinir les méthodes chargées de la création et d'accéder à certaines variables d'instance déclarées `protected`, notamment `font`, de la classe UI.

Figure 17.4

Sans changement dans le code de la classe Visualization, l'application affiche la nouvelle interface produite par la classe BetaUI.



Exercice 17.2

Suggérez un changement de conception qui permettrait toujours de développer une variété d'objets factory, mais en réduisant la dépendance des sous-classes à l'égard des modificateurs de méthodes de la classe UI.

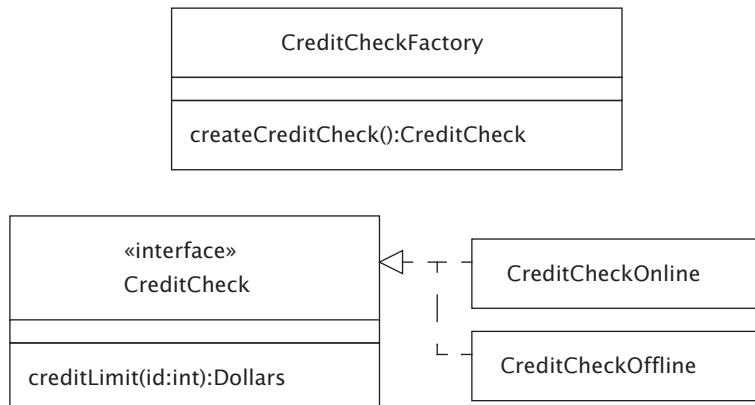
Le pattern ABSTRACT FACTORY affranchit les clients du besoin de savoir quelles classes instancier lorsqu'ils nécessitent de nouveaux objets. A cet égard, il s'apparente à un ensemble de méthodes FACTORY METHOD. Dans certains cas, une conception FACTORY METHOD peut évoluer en une conception ABSTRACT FACTORY.

Classe FACTORY abstraite et pattern FACTORY METHOD

Le Chapitre 16 a introduit une paire de classes implémentant l'interface CreditCheck. Dans la conception présentée, la classe CreditCheckFactory instancie une de ces classes lorsqu'un client appelle sa méthode createCreditCheck(), et la classe qui est instanciée dépend de la disponibilité de l'organisme de crédit. Cette conception évite aux autres développeurs d'être dépendants de cette information. La Figure 17.5 illustre la classe CreditCheckFactory et les implémentations de l'interface CreditCheck.

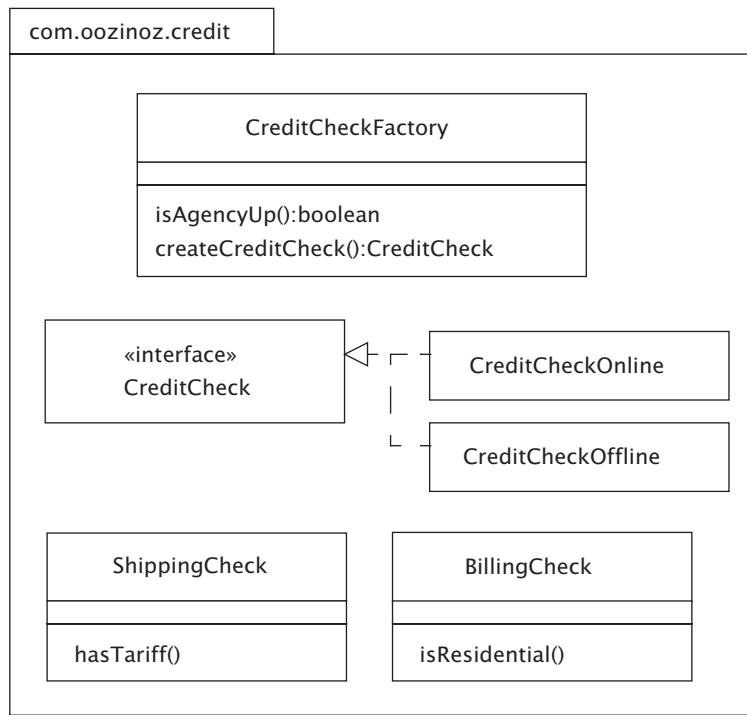
Figure 17.5

Une conception FACTORY METHOD qui exonère le code client de l'obligation de connaître la classe à instancier pour vérifier des informations de crédit.



La classe CreditCheckFactory fournit d'habitude des informations provenant de l'organisme de crédit sur la limite autorisée pour un client donné. En outre, le package credit possède des classes qui peuvent rechercher des informations d'expédition et de facturation pour un client. La Figure 17.6 illustre le package com.oozinoz.credit original.

Supposez maintenant qu'un analyste des besoins d'Oozinoz vous signale que la société est prête à prendre en charge les clients vivant au Canada. Pour travailler avec le Canada, vous utiliserez un autre organisme de crédit ainsi que d'autres sources de données pour déterminer les informations d'expédition et de facturation.

**Figure 17.6**

Les classes dans ce package vérifient le crédit d'un client, l'adresse d'expédition et l'adresse de facturation.

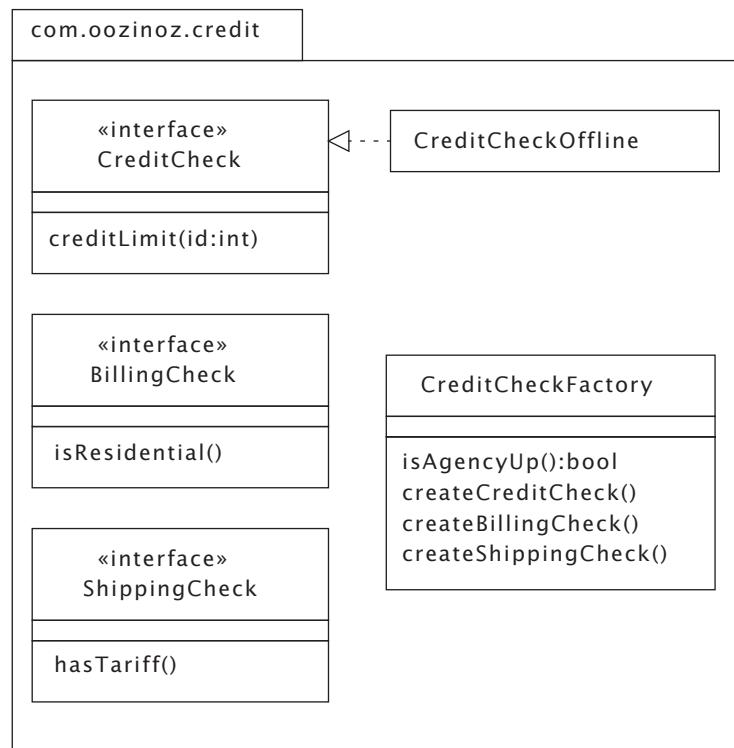
Lorsqu'un client téléphone, l'application utilisée par le centre de réception des appels doit recourir à une *famille* d'objets pour effectuer une variété de contrôles. Cette famille devra être différente selon que l'appel proviendra du Canada ou des Etats-Unis. Vous pouvez appliquer le pattern ABSTRACT FACTORY pour permettre la création de ces familles d'objets.

L'expansion de l'activité au Canada doublera pratiquement le nombre de classes sous-tendant les vérifications de crédit. Supposez que vous décidiez de coder ces classes dans trois packages. Le package `credit` contiendra maintenant trois interfaces "Check" et une classe factory abstraite. Cette classe aura trois méthodes de création pour générer les objets appropriés chargés de vérifier les informations de crédit, de facturation et d'envoi. Vous inclurez aussi la classe `CreditCheckOffline` dans ce package, partant du principe que vous pourrez l'utiliser pour effectuer

les contrôles en cas d'indisponibilité de l'organisme de crédit indépendamment de l'origine d'un appel. La Figure 17.7 montre la nouvelle composition du package com.oozinoz.credit.

Figure 17.7

Le package revu contient principalement des interfaces et une classe factory abstraite.



Pour implémenter les interfaces de `credit` avec des classes concrètes, vous pouvez introduire deux nouveaux packages : `com.oozinoz.credit.ca` et `com.oozinoz.credit.us`. Chacun de ces packages peut contenir une version concrète de la classe factory et des classes pour implémenter chacune des interfaces de `credit`.

Les classes factory concrètes pour les appels provenant du Canada et des Etats-Unis sont relativement simples. Elles retournent les versions canadiennes ou états-unien-nes des interfaces "Check", sauf si l'organisme de crédit local est hors ligne, auquel cas elles retournent toutes deux un objet `CreditCheckOffline`. Comme dans le chapitre précédent, la classe `CreditCheckFactory` possède une méthode `isAgencyUp()` qui indique si l'organisme de crédit est disponible.

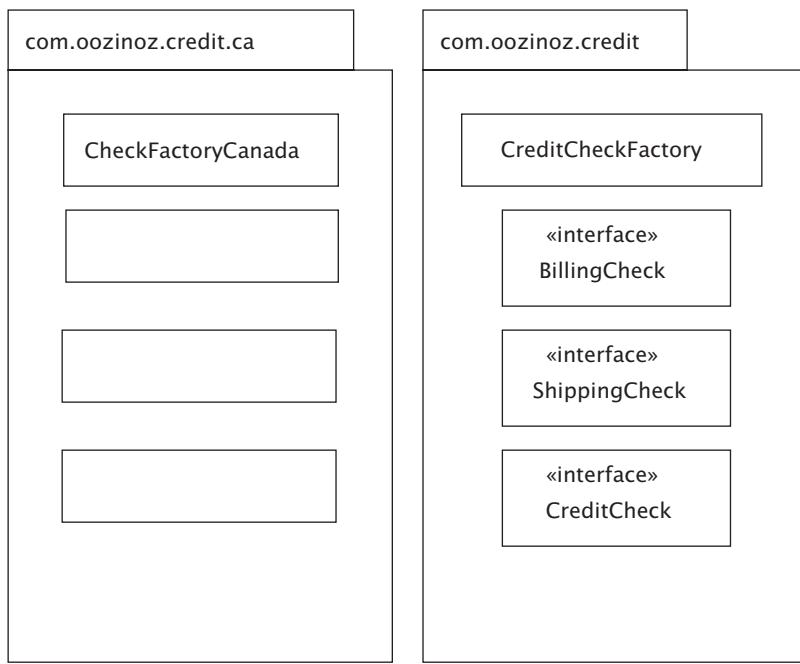


Figure 17.8

Les classes du package `com.oozinoz.credit.ca` et leurs relations avec les classes de `com.oozinoz.credit`.

Exercice 17.4

Complétez le code pour `CheckFactoryCanada.java` :

```
package com.oozinoz.credit.ca;
import com.oozinoz.credit.*;
public class CheckFactoryCanada extends CreditCheckFactory {
    // Exercice !
}
```

A ce stade, vous disposez d'une conception qui applique le pattern ABSTRACT FACTORY pour permettre la création de familles d'objets chargés de vérifier différentes informations concernant un client. Une instance de la classe `CreditCheckFactory` abstraite sera soit de la classe `CheckFactoryCanada`, soit de la classe `CheckFactoryUS`, et les objets de contrôle générés seront appropriés pour le pays représenté par l'objet factory.

Packages et classes factory abstraites

On peut quasiment dire qu'un package contient habituellement une *famille* de classes, et qu'une classe factory abstraite produit une *famille* d'objets. Dans l'exemple précédent, vous avez utilisé des packages distincts pour supporter des classes factory abstraites pour le Canada et les Etats-Unis, avec un troisième package fourniissant des interfaces communes pour les objets produits par les classes factory.

Exercice 17.5

Justifiez la décision de placer chaque classe factory et ses classes associées dans un package distinct, ou argumentez en faveur d'une autre approche jugée supérieure.

Résumé

Le pattern ABSTRACT FACTORY vous permet de prévoir la possibilité pour un client de créer des objets faisant partie d'une famille d'objets entretenant une relation. Une application classique de ce pattern concerne la création de familles de composants de GUI, ou kits. D'autres aspects peuvent aussi être traités sous forme de familles d'objets, tels que le pays de résidence d'un client. Comme pour FACTORY METHOD, ABSTRACT FACTORY vous permet d'exonérer le client de la nécessité de savoir quelle classe instancier pour créer un objet, en vous permettant de lui fournir une classe factory produisant des objets liés par un aspect commun.

PROTOTYPE

Lorsque vous développez une classe, vous prévoyez habituellement des constructeurs pour permettre aux applications clientes de l'instancier. Il y a toutefois des situations où vous souhaitez empêcher le code utilisateur de vos classes d'appeler directement un constructeur. Les patterns orientés construction décrits jusqu'à présent dans cette partie, BUILDER, FACTORY METHOD et ABSTRACT FACTORY, offrent tous la possibilité de mettre en place ce type de prévention en établissant des méthodes qui instantient une classe pour le compte du client. Le pattern PROTOTYPE dissimule également la création d'un objet mais emploie une approche différente.

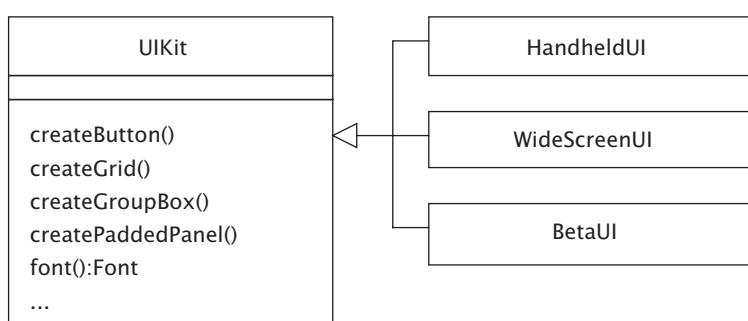
L'objectif du pattern PROTOTYPE est de fournir de nouveaux objets par la copie d'un exemple plutôt que de produire de nouvelles instances non initialisées d'une classe.

Des prototypes en tant qu'objets factory

Supposez que vous utilisez le pattern ABSTRACT FACTORY chez Oozinoz pour fournir des interfaces utilisateurs pour différents contextes. La Figure 18.1 illustre les classes factory de GUI pouvant évoluer.

Figure 18.1

Trois classes factory abstraites, ou kits, pour produire différents styles de GUI.



Les utilisateurs d’Oozinoz apprécient la productivité résultant du fait de pouvoir appliquer une GUI appropriée pour un contexte donné. Le problème que vous rencontrez est que vos utilisateurs souhaitent plusieurs kits de GUI de plus, alors qu’il devient encombrant de créer une nouvelle classe pour chaque contexte envisagé par eux. Pour stopper la prolifération des classes factory de GUI, un développeur d’Oozinoz suggère l’application du pattern **PROTOTYPE** de la manière suivante :

- supprimer les sous-classes de `UIKit` ;
- faire en sorte que chaque *instance* de `UIKit` devienne une factory de GUI qui fonctionne en générant des copies de composants prototypes ;
- placer le code qui crée de nouveaux objets `UIKit` dans des méthodes statiques de la classe `UIKit`.

Avec cette conception, un objet `UIKit` aura un jeu complet de variables prototypes d’instance : un objet bouton, un objet grille, un objet panneau avec relief de remplissage, etc. Le code qui créera un nouvel objet `UIKit` définira les valeurs des composants prototypes afin de produire l’apparence désirée. Les méthodes de création, `create()`, retourneront des copies de ces composants.

Par exemple, nous pouvons créer une méthode statique `handheldUI()` de la classe `UI`. Cette méthode instanciera `UIKit`, définira les variables d’instance avec des valeurs appropriées pour un écran d’équipement portable, et retournera l’objet à utiliser en tant que kit de GUI.

Exercice 18.1

Une conception selon **PROTOTYPE** diminuera le nombre de classes qu’Oozinoz utilise pour gérer plusieurs kits de GUI. Citez deux avantages ou inconvénients supplémentaires liés à cette approche.

- *Les solutions des exercices de ce chapitre sont données dans l’Annexe B.*

La façon normale de créer un objet est d’invoquer un constructeur d’une classe. Le pattern **PROTOTYPE** offre une solution souple, permettant de déterminer au moment de l’exécution l’objet à utiliser en tant que modèle pour le nouvel objet. Cette approche dans Java ne permet cependant pas à de nouveaux objets d’avoir des méthodes différentes de celles de leur parent. Il vous faudra donc évaluer les avantages et les inconvénients de cette technique et procéder à son expérimentation pour déterminer si elle répond à vos besoins. Pour pouvoir l’appliquer, vous devrez maîtriser les mécanismes de la copie d’objets dans Java.

Prototypage avec des clones

L'objectif du pattern PROTOTYPE est de fournir de nouveaux objets en copiant un exemple. Lorsque vous copiez un objet, le nouvel objet aura les mêmes attributs et le même comportement que ses parents. Le nouvel objet peut également hériter de certaines ou de toutes les valeurs de données de l'objet parent. Par exemple, une copie d'un panneau avec relief de remplissage devrait avoir la même valeur de remplissage que l'original.

Il est important de se demander ceci : lorsque vous copiez un objet, l'opération fournit-elle des copies des valeurs d'attributs de l'objet original ou la copie partage-t-elle ces valeurs avec l'original ? Il est facile d'oublier de se poser cette question ou d'y répondre de façon incorrecte. Les défauts apparaissent souvent lorsque les développeurs font des suppositions erronées sur les mécanismes de la copie. De nombreuses classes dans les bibliothèques de classes Java offrent un support pour la copie, mais en tant que développeur, vous devez comprendre comment la copie fonctionne, surtout si vous voulez utiliser PROTOTYPE.

Exercice 18.2

La classe `Object` inclut une méthode `clone()` dont tous les objets héritent. Si cette méthode ne vous est pas familière, reportez-vous à l'aide en ligne ou à la documentation. Ecrivez ensuite dans vos propres termes ce que cette méthode effectue.

Exercice 18.3

Supposez que la classe `Machine` possédait deux attributs : un entier `ID` et un emplacement, `Location`, sous forme d'une classe distincte.

Dessinez un diagramme objet montrant un objet `Machine`, son objet `Location`, et tout autre objet résultant de l'appel de `clone()` sur l'objet `Machine`.

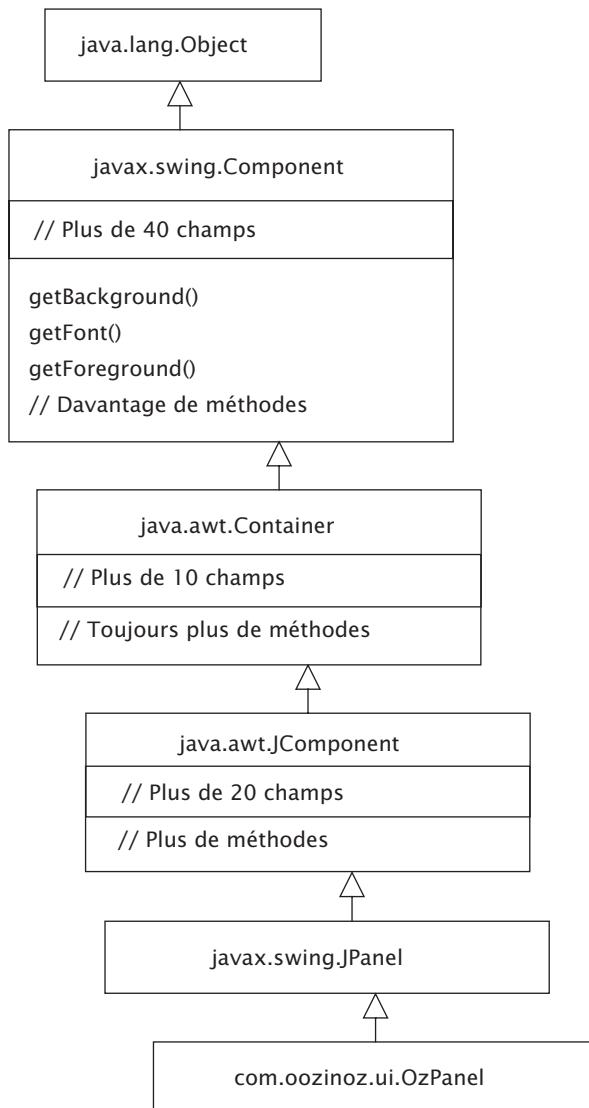
La méthode `clone()` facilite l'ajout d'une méthode `copy()` à une classe. Par exemple, vous pourriez créer une classe de panneaux pouvant être clonés au moyen du code suivant :

```
package com.oozinoz.ui;
import javax.swing.JPanel;
```

```
public class OzPanel extends JPanel implements Cloneable {  
    // Dangereux !  
    public OzPanel copy() {  
        return (OzPanel) this.clone();  
    }  
    // ...  
}
```

Figure 18.2

La classe OzPanel hérite d'un grand nombre de champs et de variables de ses super-classes.



La méthode `copy()` dans ce code rend le clonage public et convertit la copie dans le type adéquat. Le problème de ce code est que la méthode `clone()` créera des copies de tous les attributs d'un objet `JPanel1`, indépendamment du fait que vous comprenez ou non la fonction de ces attributs. Notez que les attributs de la classe `JPanel1` incluent les attributs des classes ancêtres, comme le montre la Figure 18.2.

Comme le suggère la Figure 18.2, la classe `OzPanel1` hérite d'un nombre important de propriétés de la classe `Component`, et ce sont souvent les seuls attributs qu'il vous faut copier lorsque vous travaillez avec des objets de GUI.

Exercice 18.4

Ecrivez une méthode `OzPanel1.copy2()` qui copie un panneau sans s'appuyer sur `clone()`. Supposez que les seuls attributs importants pour une copie sont `background`, `font` et `foreground`.

Résumé

Le pattern PROTOTYPE permet à un client de créer de nouveaux objets en copiant un exemple. Une grande différence entre appeler un constructeur et copier un objet est qu'une copie inclut généralement un certain état de l'objet original. Vous pouvez utiliser cela à votre avantage, surtout lorsque différentes catégories d'objets ne diffèrent que par leurs attributs et non dans leurs comportements. Dans ce cas, vous pouvez créer de nouvelles classes au moment de l'exécution en générant des objets prototypes que le client peut copier.

Lorsque vous devez créer une copie, la méthode `Object.clone()` peut être utile, mais vous devez vous rappeler qu'elle crée un nouvel objet avec les mêmes champs. Cet objet peut ne pas être une copie convenable, et toute difficulté liée à une opération de copie plus importante relève de votre responsabilité. Si un objet prototype possède trop de champs, vous pouvez créer un nouvel objet par instantiation et en définissant ses champs de manière à ne représenter que les aspects de l'objet original que vous voulez copier.

MEMENTO

Il y a des situations où l'objet que vous voulez créer existe déjà. Cela se produit lorsque vous voulez laisser un utilisateur annuler des opérations, revenir à une version précédente d'un travail, ou reprendre un travail suspendu.

L'objectif du pattern MEMENTO est de permettre le stockage et la restauration de l'état d'un objet.

Un exemple classique : défaire une opération

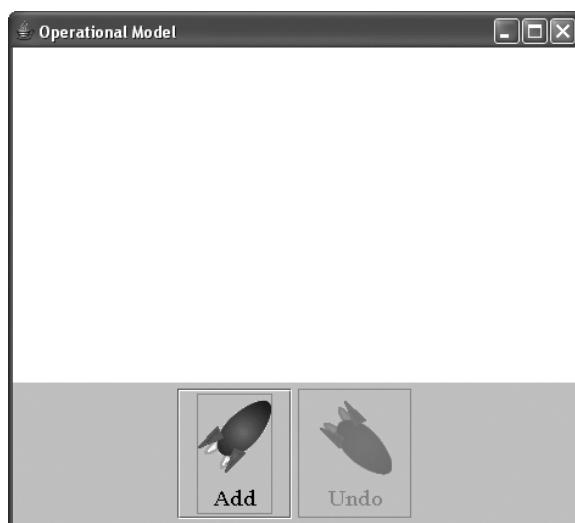
Le Chapitre 17 a introduit une application de visualisation permettant à ses utilisateurs d'expérimenter la modélisation des flux matériels dans une usine. Supposez que la fonctionnalité du bouton Undo n'ait pas encore été implémentée. Nous pouvons appliquer le pattern MEMENTO pour faire fonctionner ce bouton.

Un objet **memento** contient des informations d'état. Dans l'application de visualisation, l'état que nous devons préserver est celui de l'application. Lors de l'ajout ou du déplacement d'une machine, un utilisateur devrait être en mesure d'annuler l'opération en cliquant sur le bouton Undo. Pour ajouter cette fonctionnalité, nous devons décider de la façon de capturer l'état de l'application dans un objet memento. Nous devrons aussi décider du moment auquel le faire, et comment le restaurer au besoin. Lorsque l'application démarre, elle apparaît comme illustré Figure 19.1.

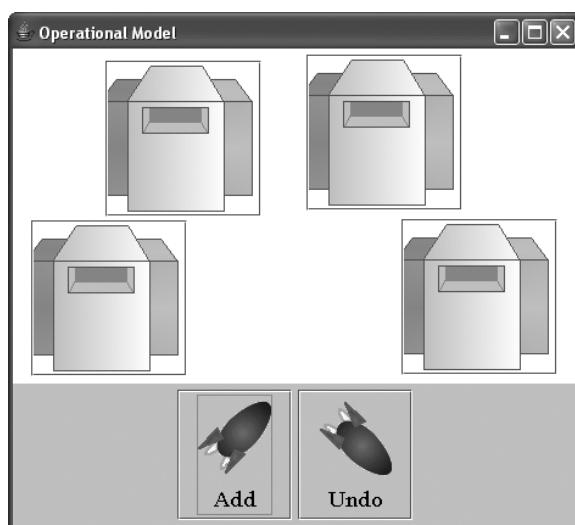
L'application démarre vierge, ce qui *est* malgré tout un état. Dans ce cas, le bouton Undo devrait être désactivé. Après quelques ajouts et déplacements, la fenêtre pourrait ressembler à l'exemple de la Figure 19.2.

Figure 19.1

Lorsque l'application démarre, le panneau est vierge et le bouton Undo est désactivé.

**Figure 19.2**

L'application après quelques ajouts et positionnements de machines.



L'état qu'il nous faut enregistrer dans un memento est une liste des emplacements des machines qui ont été placées par l'utilisateur. Nous pouvons empiler ces mémentos, en en dépiler à chaque fois que l'utilisateur clique sur le bouton Undo :

- Chaque fois que l'utilisateur ajoute ou déplace une machine, le code doit créer un memento du factory simulé et l'ajouter à une pile.

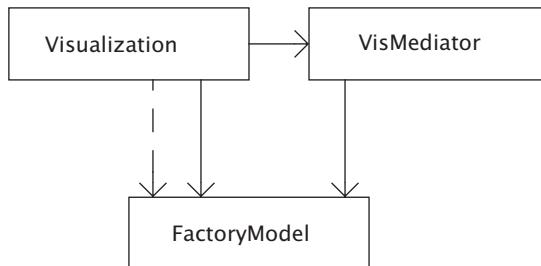
- Chaque fois qu'il clique sur le bouton Undo, le code doit retirer le memento le plus récent, le plus haut dans la pile, et restaurer la simulation dans l'état qui y aura été enregistré.

Lorsque l'application démarre, vous empilez un memento vide qui n'est jamais prélevé pour garantir que le sommet de la pile sera toujours un memento valide. Chaque fois que la pile ne contient qu'un memento, vous désactivez le bouton Undo.

Nous pourrions écrire le code de ce programme dans une seule classe, mais nous envisageons l'ajout de fonctionnalités pour gérer la modélisation opérationnelle et d'autres fonctions que les utilisateurs pourront éventuellement demander. Finalement, l'application pouvant devenir plus grande, il est sage de s'appuyer sur une conception MVC (Modèle-Vue-Contrôleur). La Figure 19.3 illustre une conception qui place le travail de modélisation de l'objet factory en classes distinctes.

Figure 19.3

Cette conception divise le travail de l'application en classes distinctes, pour modéliser l'objet factory, fournir les éléments de GUI et gérer les clics de l'utilisateur.



Cette conception vous permet de vous concentrer d'abord sur le développement d'une classe `FactoryModel` qui ne possède pas de composants de GUI et aucune dépendance à l'égard de la GUI.

La classe `FactoryModel` est au cœur de la conception. Elle est responsable de la gestion de la configuration actuelle des machines et des mémentos des configurations antérieures.

Chaque fois qu'un client demande à l'objet factory d'ajouter ou de déplacer une machine, celui-ci crée une copie, un objet memento, de l'emplacement actuel des machines, et place l'objet sur la pile de mémentos. Dans cet exemple, nous n'avons pas besoin d'une classe `Memento` spéciale. Chaque memento est simplement une liste de points : la liste des emplacements de l'équipement à un moment donné.

Le modèle de conception de l'usine doit prévoir des événements pour permettre aux clients de s'enregistrer pour signaler leur intérêt à connaître les changements d'état de l'usine. Cela permet à la GUI d'informer le modèle de changements que l'utilisateur effectue. Supposez que vous vouliez que le factory laisse les clients s'enregistrer pour connaître les événements d'ajout et de déplacement de machine. La Figure 19.4 illustre une conception pour une classe `FactoryModel`.

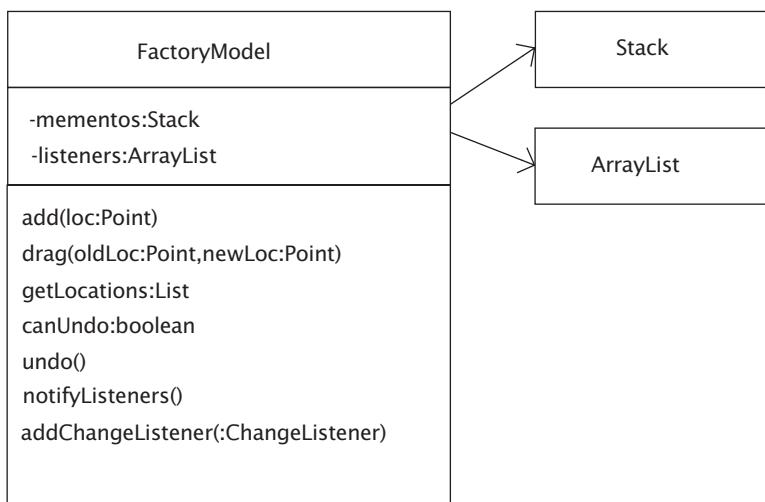


Figure 19.4

La classe `FactoryModel` conserve une pile de configurations matérielles et permet aux clients de s'enregistrer pour être notifiés des changements intervenant dans l'usine.

La conception de la Figure 19.4 prévoit que la classe `FactoryModel` donne aux clients la possibilité de s'enregistrer pour être notifiés de plusieurs événements.

Par exemple, considérez l'événement d'ajout d'une machine. Tout objet `ChangeListener` enregistré sera notifié de ce changement :

```

package com.oozinoz.visualization;
// ...
public class FactoryModel {
    private Stack mementos;

    private ArrayList listeners = new ArrayList();

    public FactoryModel() {
  
```

```
        mementos = new Stack();
        mementos.push(new ArrayList());
    }
    //...
}
```

Le constructeur débute la configuration initiale de l'usine sous forme d'une liste vierge. Les autres méthodes de la classe gèrent la pile des mémentos et déclenchent les événements qui correspondent à tout changement. Par exemple, pour ajouter une machine à la configuration actuelle, un client peut appeler la méthode suivante :

```
public void add(Point location) {
    List oldLocs = (List) mementos.peek();
    List newLocs = new ArrayList(oldLocs);
    newLocs.add(0, location);
    mementos.push(newLocs);
    notifyListeners();
}
```

Ce code crée une nouvelle liste d'emplacements des machines et la place sur la pile gérée par le modèle. Une subtilité du code est de s'assurer que la nouvelle machine soit d'abord dans cette liste. C'est un signe pour la visualisation qu'elle doit alors apparaître devant les autres machines que l'affichage pourrait faire se chevaucher.

Un client qui s'enregistre pour recevoir les notifications de changements pourrait actualiser la vue de son modèle en se reconstruisant lui-même entièrement à la réception d'un événement de la part du modèle. La configuration la plus récente du modèle est toujours disponible dans `getLocations()`, dont le code se présente ainsi :

```
public List getLocations() {
    return (List) mementos.peek();
}
```

La méthode `undo()` de la classe `FactoryModel` permet à un client de changer le modèle de positionnement de machines pour restituer une version précédente. Lorsque ce code s'exécute, il invoque aussi `notifyListeners()`.

Exercice 19.1

Ecrivez le code de la méthode `undo()` de la classe `FactoryModel`.

- *Les solutions des exercices de ce chapitre sont données dans l'Annexe B.*

Un client intéressé peut fournir une fonctionnalité d'annulation d'opérations en enregistrant un listener et en fournissant une méthode qui reconstruit la vue du modèle. La classe `Visualization` est un client de ce type.

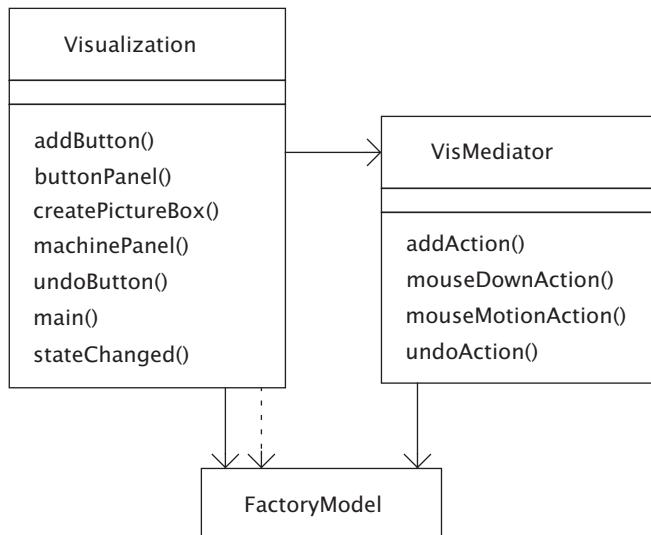
La conception MVC illustrée à la Figure 19.3 sépare les tâches d'interprétation des actions de l'utilisateur de celles de gestion de la GUI. La classe `Visualization` crée ses composants de GUI mais fait passer la gestion des événements de GUI à un médiateur. La classe `VisMediator` traduit les événements de GUI en changements appropriés dans le modèle. Lorsque celui-ci change, la GUI peut nécessiter une actualisation. La classe `Visualization` s'enregistre pour recevoir les notifications fournies par la classe `FactoryModel`. Notez la séparation des responsabilités.

- La visualisation change les événements du modèle en changements de la GUI.
- Le médiateur traduit les événements de GUI en changements du modèle.

La Figure 19.5 illustre en détail les trois classes qui collaborent.

Figure 19.5

Le médiateur traduit les événements de GUI en changements du modèle, et la visualisation réagit aux événements de changements du modèle pour actualiser la GUI.



Supposez que, pendant le déplacement d'une image de machine, un utilisateur la dépose accidentellement au mauvais endroit et clique sur le bouton Undo. Pour pouvoir gérer ce clic, la visualisation enregistre le médiateur pour qu'il reçoive les

notifications d'événements de bouton. Le code du bouton Undo dans la classe `Visualization` se présente comme suit :

```
protected JButton undoButton() {
    if (undoButton == null) {
        undoButton = ui.createButtonCancel();
        undoButton.setText("Undo");
        undoButton.setEnabled(false);
        undoButton.addActionListener(mediator.undoAction());
    }
    return undoButton;
}
```

Ce code délègue au médiateur la responsabilité de la gestion d'un clic. Le médiateur informe le modèle de tout changement demandé et traduit une requête d'annulation d'opération en un changement du modèle au moyen du code suivant :

```
private void undo(ActionEvent e) {
    factoryModel.undo();
}
```

La variable `factoryModel` dans cette méthode est une instance de `FactoryModel`, que la classe `Visualization` crée et passe au médiateur *via* le constructeur de la classe `VisMediator`. Nous avons déjà examiné la commande `pop()` de la classe `FactoryModel`. Le flux de messages qui est généré lorsque l'utilisateur clique sur Undo est présenté Figure 19.6.

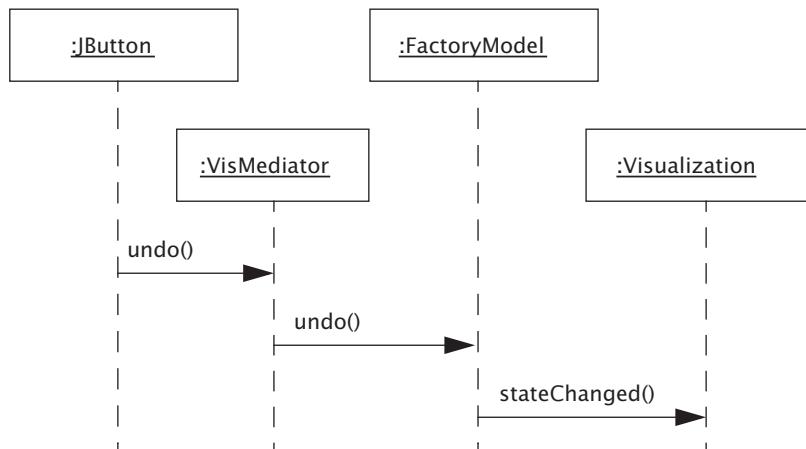


Figure 19.6

Le diagramme illustre les messages générés suite à l'activation du bouton Undo.

Lorsque la classe `FactoryModel` prélève de la pile la configuration précédente qu'elle a stockée en tant que memento, la méthode `undo()` notifie les `ChangeListeners`. La classe `Visualization` a prévu son enregistrement à cet effet dans son constructeur :

```
public Visualization(UI ui) {  
    super(new BorderLayout());  
    this.ui = ui;  
    mediator = new VisMediator(factoryModel);  
    factoryModel.addChangeListener(this);  
    add(machinePanel(), BorderLayout.CENTER);  
    add(buttonPanel(), BorderLayout.SOUTH);  
}
```

Pour chaque position de machine dans le modèle, la visualisation conserve un objet `Component` qu'il crée avec la méthode `createPictureBox()`. La méthode `stateChanged()` doit nettoyer tous les composants en place dans le panneau et rétablir les encadrés des positions restaurées. La méthode `stateChanged()` doit aussi désactiver le bouton Undo s'il ne reste qu'un memento sur la pile.

Exercice 19.2

Ecrivez la méthode `stateChanged()` pour la classe `Visualization`.

Le pattern MEMENTO permet de sauvegarder et de restaurer l'état d'un objet. Une application courante de ce pattern est la gestion de la fonctionnalité d'annulation d'opérations dans les applications. Dans certaines applications, comme dans l'exemple de visualisation des machines de l'usine, l'entrepôt où stocker les informations sauvegardées peut être un autre objet. Dans d'autres cas, les mémentos peuvent être stockés sous une forme plus durable.

Durée de vie des mémentos

Un memento est un petit entrepôt qui conserve l'état d'un objet. Vous pouvez créer un memento en utilisant un autre objet, une chaîne ou un fichier. La durée anticipée entre le stockage et la reconstruction d'un objet a un impact sur la stratégie que vous utilisez dans la conception d'un memento. Il peut s'agir d'un court instant, mais aussi d'heures, de jours ou d'années.

Exercice 19.3

Indiquez deux raisons qui peuvent motiver l'enregistrement d'un memento dans un fichier plutôt que sous forme d'objet.

Persistance des mémentos entre les sessions

Une **session** se produit lorsqu'un utilisateur exécute un programme, réalise des transactions par son intermédiaire, puis le quitte. Supposez que vos utilisateurs souhaitent pouvoir sauvegarder une simulation d'une session et la restaurer dans une autre session. Cette fonctionnalité est un concept normalement appelé **stockage persistant**. Le stockage persistant satisfait l'objectif du pattern MEMENTO et constitue une extension naturelle de la fonctionnalité d'annulation que nous avons déjà implémentée.

Supposez que vous dériviez une sous-classe `Visualization2` de la classe `Visualization`, qui possède une barre de menus avec un menu File comportant les options Save As... et Restore From... :

```
package com.oozinoz.visualization;

import javax.swing.*;
import com.oozinoz.ui.SwingFacade;
import com.oozinoz.ui.UI;

public class Visualization2 extends Visualization {
    public Visualization2(UI ui) {
        super(ui);
    }

    public JMenuBar menus() {
        JMenuBar menuBar = new JMenuBar();

        JMenu menu = new JMenu("File");
        menuBar.add(menu);

        JMenuItem menuItem = new JMenuItem("Save As...");
        menuItem.addActionListener(mediator.saveAction());
        menu.add(menuItem);

        menuItem = new JMenuItem("Restore From...");
        menuItem.addActionListener(
            mediator.restoreAction());
        menu.add(menuItem);
        return menuBar;
    }
}
```

```

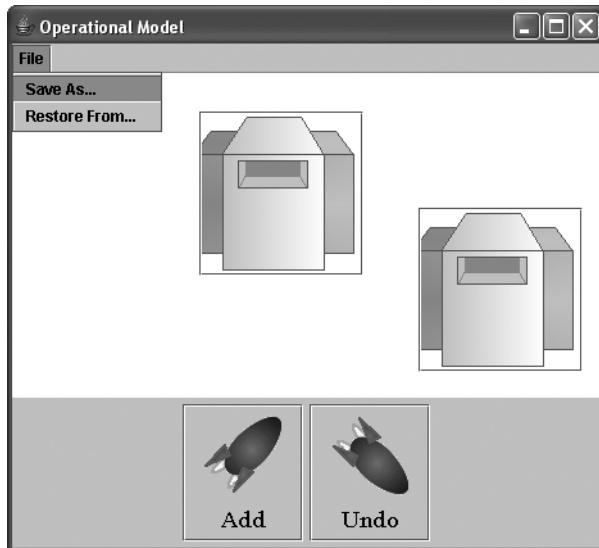
public static void main(String[] args) {
    Visualization2 panel = new Visualization2(UI.NORMAL);
    JFrame frame = SwingFacade.launch(
        panel, "Operational Model");
    frame.setJMenuBar(panel.menus());
    frame.setVisible(true);
}
}

```

Ce code requiert l'ajout des méthodes `saveAction()` et `restoreAction()` à la classe `VisMediator`. Les objets `MenuItem` provoquent l'appel de ces actions lorsque le menu est sélectionné. Lorsque la classe `Visualization2` s'exécute, la GUI se présente comme illustré Figure 19.7.

Figure 19.7

L'ajout d'un menu permet à l'utilisateur d'enregistrer un memento que l'application pourra restaurer lors d'une prochaine session.



Un moyen facile de stocker un objet, telle la configuration du modèle de visualisation, est de le sérialiser. Le code de la méthode `saveAction()` dans la classe `VisMediator` pourrait être comme suit :

```

public ActionListener saveAction() {
    return new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            try {
                VisMediator.this.save((Component)e.getSource());
            }

```

```
        } catch (Exception ex) {
            System.out.println(
                "Echec de sauvegarde : " + ex.getMessage());
        }
    };
}

public void save(Component source) throws Exception {
    JFileChooser dialog = new JFileChooser();
    dialog.showSaveDialog(source);

    if (dialog.getSelectedFile() == null)
        return;

    FileOutputStream out = null;
    ObjectOutputStream s = null;
    try {
        out = new FileOutputStream(dialog.getSelectedFile());
        s = new ObjectOutputStream(out);
        s.writeObject(factoryModel.getLocations());
    } finally {
        if (s != null) s.close();
    }
}
```

Exercice 19.4

Ecrivez le code de la méthode `restoreAction()` de la classe `VisMediator`.

L'ouvrage *Design Patterns* décrit ainsi l'objectif du pattern MEMENTO : "Sans enfreindre les règles d'encapsulation, il capture et externalise l'état interne d'un objet afin de pouvoir le restaurer ultérieurement."

Exercice 19.5

Dans ce cas, nous avons utilisé la sérialisation Java pour enregistrer la configuration dans un fichier au format binaire. Supposez que nous l'ayons enregistré dans le format XML (texte). Expliquez brièvement pourquoi, à votre avis, l'enregistrement d'un memento au format texte serait une atteinte à la règle d'encapsulation.

Vous devriez comprendre ce qu'un développeur signifie lorsqu'il indique qu'il crée des mémentos en stockant les données d'objets au moyen de la sérialisation ou de l'enregistrement dans un fichier XML. C'est l'idée des patterns de conception : en utilisant un vocabulaire commun, nous pouvons discuter de concepts de conception et de leurs applications.

Résumé

Le pattern MEMENTO permet de capturer l'état d'un objet de manière à pouvoir le restaurer ultérieurement. La méthode de stockage utilisée à cet effet dépend du type de restauration à faire, après un clic ou une frappe au clavier ou lors d'une prochaine session après un certain temps. La raison la plus courante de réaliser cela est toutefois de supporter la fonction d'annulation d'actions précédentes dans une application. Dans ce cas, vous pouvez stocker l'état d'un objet dans un autre objet. Pour que le stockage de l'état soit persistant, vous pouvez utiliser, entre autres moyens, la sérialisation d'objet.

IV

Patterns d'opération

Introduction aux opérations

Lorsque vous écrivez une méthode Java, vous produisez une unité fondamentale de traitement qui intervient un niveau au-dessus de celui de l'écriture d'une instruction. Vos méthodes doivent participer à une conception, une architecture et un plan de test d'ensemble, et en même temps l'écriture de méthodes est au cœur de la POO. En dépit de ce rôle central, une certaine confusion règne quant à ce que les méthodes sont vraiment et comment elles fonctionnent, qui vient probablement du fait que les développeurs — et les auteurs — ont tendance à utiliser de façon interchangeable les termes *méthode* et *opération*. De plus, les concepts d'*algorithme* et de *polymorphisme*, bien que plus abstraits que les méthodes, sont au final mis en œuvre par elles.

Une définition claire des termes *algorithme*, *polymorphisme*, *méthode* et *opération* vous aidera à comprendre plusieurs patterns de conception. En particulier, STATE, STRATEGY et INTERPRETER fonctionnent tous trois en implémentant une opération dans des méthodes à travers plusieurs classes, mais une telle observation n'a d'utilité que si nous nous entendons sur le sens de *méthode* et d'*opération*.

Opérations et méthodes

Parmi les nombreux termes relatifs au traitement qu'une classe peut être amenée à effectuer, il est particulièrement utile de distinguer une *opération* d'une *méthode*. Le langage UML définit cette différence comme suit :

- Une **opération** est la spécification d'un service qui peut être demandé par une instance d'une classe.
- Une **méthode** est l'implémentation d'une opération.

Notez que la signification d'**opération** se situe un niveau d'abstraction au-dessus de la notion de **méthode**.

Une opération spécifie quelque chose qu'une classe accomplit et spécifie l'interface pour appeler ce service. Plusieurs classes peuvent implémenter la même opération de différentes manières. Par exemple, nombre de classes implémentent l'opération `toString()` chacune à sa façon. Chaque classe qui implémente une opération utilise pour cela une méthode. Cette méthode contient — ou est — le code qui permet à l'opération de fonctionner pour cette classe.

Les définitions de *méthode* et *opération* permettent de clarifier la structure de nombreux patterns de conception. Etant donné que ces patterns se situent un niveau au-dessus des classes et des méthodes, il n'est pas surprenant que les opérations soient prédominantes dans de nombreux patterns. Par exemple, **COMPOSITE** permet d'appliquer des opérations à la fois à des éléments et à des groupes, et **PROXY** permet à un intermédiaire implémentant les mêmes opérations qu'un objet cible de s'interposer pour gérer l'accès à cet objet.

Exercice 20.1

Utilisez les termes **opération** et **méthode** pour expliquer comment le pattern **CHAIN OF RESPONSIBILITY** implémente une opération.

- *Les solutions des exercices de ce chapitre sont données dans l'Annexe B.*

Dans Java, la déclaration d'une méthode inclut un *en-tête (header)* et un *corps (body)*. Le **corps** est la série d'instructions qui peuvent être exécutées en invoquant la signature de la méthode. L'**en-tête** inclut le type de retour et la signature de la méthode et peut aussi inclure des modificateurs et une clause `throws`. Voici le format de l'en-tête d'une méthode :

modificateurs type-retour signature clause-throws

Exercice 20.2

Parmi les neuf modificateurs de méthodes Java, énumérez tous ceux que vous pouvez.

Signatures

En surface, la signification d'**opération** est semblable à celle de **signature**, ces deux termes désignant l'interface d'une méthode. Lorsque vous écrivez une méthode, elle devient invocable conformément à sa signature. La Section 8.4.2 de l'ouvrage *JavaTM Language Specification* [Arnold et Gosling 1998] donne la définition suivante d'une signature :

La signature d'une méthode est constituée du nom de la méthode ainsi que du nombre et des types de paramètres formels qu'elle reçoit.

Notez que la signature d'une méthode n'inclut pas son type de retour. Toutefois, si la déclaration d'une méthode remplace la déclaration d'une autre méthode, une erreur de compilation surviendra si elles possèdent des types de retour différents.

Exercice 20.3

La méthode `Bitmap.clone()` retourne toujours une instance de la classe `Bitmap` bien que son type de retour soit `Object`. Serait-elle compilée sans erreur si son type de retour était `Bitmap` ?

Une signature spécifie quelle méthode est invoquée lorsqu'un client effectue un appel. Une *opération* est la spécification d'un service qui peut être demandé. Les termes *signature* et *opération* ont une signification analogue, bien qu'ils ne soient pas synonymes. La différence a trait principalement au contexte dans lequel ils s'appliquent. Le terme *opération* est employé en rapport avec l'idée que des méthodes de différentes classes peuvent avoir la même interface. Le terme *signature* est employé en rapport avec les règles qui déterminent comment Java fait correspondre l'appel d'une méthode à une méthode de l'objet récepteur. Une signature dépend du nom et des paramètres d'une méthode mais pas du type de retour de celle-ci.

Exceptions

Dans son livre *La maladie comme métaphore*, Susan Sontag observe ceci : "En naissant, nous acquérons une double nationalité qui relève du royaume des bien-portants comme de celui des malades." Cette métaphore peut aussi s'appliquer aux méthodes : au lieu de se terminer normalement, une méthode peut générer une

exception ou invoquer une autre méthode pour cela. Lorsqu'une méthode se termine normalement, le contrôle du programme revient au point situé juste après l'appel. Un autre ensemble de règles s'appliquent dans le royaume des exceptions.

Lorsqu'une exception est générée, l'environnement d'exécution Java doit trouver une instruction `try/catch` correspondante. Cette instruction peut exister dans la méthode qui a généré l'exception, dans la méthode qui a appelé la méthode courante, ou dans la méthode qui a appelé la méthode précédente, et ainsi de suite en remontant la pile d'appels. En l'absence d'instruction `try/catch` correspondante, le programme plante.

N'importe quelle méthode peut générer une exception en utilisant une instruction `throw`. Par exemple :

```
throw new Exception("Bonne chance !");
```

Si votre application utilise une méthode qui génère une exception que vous n'avez pas prévue, cela peut la faire planter. Pour éviter ce genre de comportement, vous devez disposer d'un plan architectural qui spécifie les points dans votre application où les exceptions sont interceptées et gérées de façon appropriée. Vous pensez probablement qu'il n'est pas très commode d'avoir à déclarer l'éventualité d'une exception. Dans C#, par exemple, les méthodes n'ont pas besoin de déclarer des exceptions. Dans C++, une exception peut apparaître sans que le compilateur doive vérifier qu'elle a été prévue par les appelsants.

Exercice 20.4

Contrairement à Java, C# n'impose pas aux méthodes de déclarer les exceptions qu'elles peuvent être amenées à générer. Pensez-vous que Java constitue une amélioration à cet égard ? Expliquez votre réponse.

Algorithmes et polymorphisme

Les algorithmes et le polymorphisme sont deux concepts importants en programmation, mais il peut être difficile de donner une explication de ces termes. Si vous voulez montrer à quelqu'un une méthode, vous pouvez modifier le code source d'une classe en mettant en évidence les lignes de code appropriées. Parfois, un algorithme peut exister entièrement dans une méthode, mais il s'appuie le plus souvent sur l'interaction de plusieurs méthodes. L'ouvrage *Introduction to*

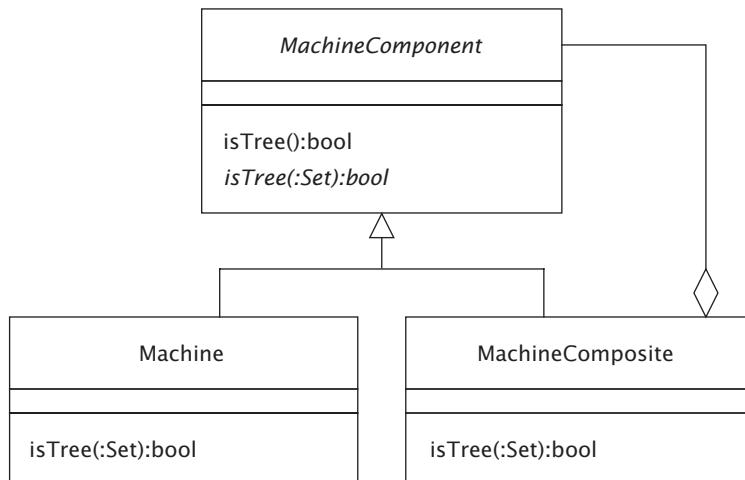
Algorithms (Introduction à l’algorithmique) [Cormen, Leiserson, et Rivest 1990, p. 1] affirme ceci :

Un *algorithme* est une procédure de calcul bien définie qui reçoit une ou plusieurs valeurs en entrée et produit une ou plusieurs valeurs en sortie.

Un **algorithme** est une procédure, c'est-à-dire une séquence d'instructions, qui accepte une entrée et produit un résultat. Comme il a été dit, une seule méthode peut constituer un algorithme : elle accepte une entrée — sa liste de paramètres — et produit sa valeur de retour en sortie. Toutefois, nombre d'algorithmes requièrent plusieurs méthodes pour s'exécuter dans un programme orienté objet. Par exemple, l'algorithme `isTree()` du Chapitre 5, consacré au pattern COMPOSITE, nécessite quatre méthodes, comme le montre la Figure 20.1.

Figure 20.1

Quatre méthodes `isTree()` forment l'algorithme et collaborent pour déterminer si une instance de `MachineComponent` est un arbre.



Exercice 20.5

Combien d'algorithmes, d'opérations et de méthodes la Figure 20.1 comprend-elle ?

Un algorithme réalise un traitement. Il peut être contenu dans une seule méthode ou bien nécessiter de nombreuses méthodes. En POO, les algorithmes qui requièrent plusieurs méthodes s'appuient souvent sur le *polymorphisme* pour autoriser

plusieurs implémentations d'une même opération. Le **polymorphisme** est le principe selon lequel la méthode appelée dépend à la fois de l'opération invoquée et de la classe du récepteur de l'appel. Par exemple, vous pourriez vous demander quelle méthode est exécutée lorsque Java rencontre l'expression `m.isTree()`. Cela dépend. Si `m` est une instance de `Machine`, Java invoquera `Machine.isTree()`. S'il s'agit d'une instance de `MachineComposite`, il invoquera `MachineComposite.isTree()`. De manière informelle, le polymorphisme signifie que la méthode appropriée sera invoquée pour le type d'objet approprié. Nombre de patterns emploient le principe de polymorphisme, lequel est parfois directement lié à l'objectif du pattern.

Résumé

Même si les termes *opération*, *méthode*, *signature* et *algorithme* semblent souvent avoir une signification proche, préserver leur distinction facilite la description de concepts importants. A l'instar d'une signature, une opération est la spécification d'un service. Le terme *opération* est employé en rapport avec l'idée que de nombreuses méthodes peuvent avoir la même interface. Le terme *signature* est employé en rapport avec les règles de recherche de la méthode appropriée. La définition d'une méthode inclut sa signature, c'est-à-dire son nom et sa liste de paramètres, ainsi que des modificateurs, un type de retour et le corps de la méthode. Une méthode possède une signature et implémente une opération.

La voie normale d'invocation d'une méthode consiste à l'appeler. Une méthode doit normalement se terminer en retournant une valeur, mais interrompra son exécution si elle rencontre une exception non gérée.

Un *algorithme* est une procédure qui accepte une entrée et produit un résultat. Une méthode accepte elle aussi une entrée et produit un résultat, et comme elle contient un corps procédural, certains auteurs qualifient ce corps d'*algorithme*. Mais étant donné que la procédure algorithmique peut faire intervenir de nombreuses opérations et méthodes, ou peut exister au sein d'une même méthode, il est plus correct de réservier le terme *algorithme* pour désigner une procédure produisant un résultat.

Beaucoup de patterns de conception impliquent la distribution d'une opération à travers plusieurs classes. On peut également dire que ces patterns s'appuient sur le polymorphisme, principe selon lequel la sélection d'une méthode dépend de la classe de l'objet qui reçoit l'appel.

Au-delà des opérations ordinaires

Différentes classes peuvent implémenter une opération de différentes manières. Autrement dit, Java supporte le polymorphisme. La puissance de ce concept pourtant simple apparaît dans plusieurs patterns de conception.

<i>Si vous envisagez de</i>	<i>Appliquez le pattern</i>
• Implémenter un algorithme dans une méthode, remettant à plus tard la définition de certaines étapes de l'algorithme pour permettre à des sous-classes de les redéfinir	TEMPLATE METHOD
• Distribuer une opération afin que chaque classe représente un état différent	STATE
• Encapsuler une opération, rendant les implémentations interchangeables	STRATEGY
• Encapsuler un appel de méthode dans un objet	COMMAND
• Distribuer une opération de façon que chaque implémentation s'applique à un type différent de composition	INTERPRETER

Les patterns d'opération conviennent dans des contextes où vous avez besoin de plusieurs méthodes, généralement avec la même signature, pour participer à une conception. Par exemple, le pattern TEMPLATE METHOD permet à des sous-classes d'implémenter des méthodes qui ajustent l'effet d'une procédure définie dans une super-classe.

TEMPLATE METHOD

Les méthodes ordinaires ont un corps qui définit une séquence d'instructions. Il est également assez commun pour une méthode d'invoquer des méthodes de l'objet courant et d'autres objets. Dans ce sens, les méthodes ordinaires sont des "modèles" (*template*) qui exposent une série d'instructions que l'ordinateur doit suivre. Le pattern TEMPLATE METHOD implique un type plus spécifique de modèle.

Lorsque vous écrivez une méthode, vous pouvez vouloir définir la structure générale d'un algorithme tout en laissant la possibilité d'implémenter différemment certaines étapes. Dans ce cas, vous pouvez définir la méthode mais faire de ces étapes des méthodes abstraites, des méthodes stub, ou des méthodes définies dans une interface séparée. Cela produit un modèle plus rigide qui définit spécifiquement quelles étapes d'un algorithme peuvent ou doivent être fournies par d'autres classes.

L'objectif du pattern TEMPLATE METHOD est d'implémenter un algorithme dans une méthode, laissant à d'autres classes le soin de définir certaines étapes de l'algorithme.

Un exemple classique : algorithme de tri

Les algorithmes de tri ne datent pas d'hier et sont hautement réutilisables. Imaginez qu'un homme préhistorique ait élaboré une méthode pour trier des flèches en fonction du degré d'affûtage de leur tête. La méthode consiste à aligner les flèches puis à effectuer une série de permutations gauche-droite, remplaçant chaque flèche par

une flèche plus affûtée située sur sa gauche. Cet homme pourrait ensuite appliquer la même méthode pour trier les flèches selon leur portée ou tout autre critère.

Les algorithmes de tri varient en termes d'approche et de rapidité, mais tous se fondent sur le principe primitif de comparaison de deux éléments ou attributs. Si vous disposez d'un algorithme de tri et pouvez comparer un certain attribut de deux éléments, l'algorithme vous permettra d'obtenir une collection d'éléments triés d'après cet attribut.

Le tri est un exemple de **TEMPLATE METHOD**. Il s'agit d'une procédure qui nous permet de modifier une étape critique, à savoir la comparaison de deux objets, afin de pouvoir réutiliser l'algorithme pour divers attributs de différentes collections d'objets.

A notre époque, le tri est probablement l'algorithme le plus fréquemment réimplémenté, le nombre d'implémentations dépassant vraisemblablement le nombre de programmeurs. Mais à moins d'avoir à trier une énorme collection, vous n'avez généralement pas besoin d'écrire votre propre algorithme.

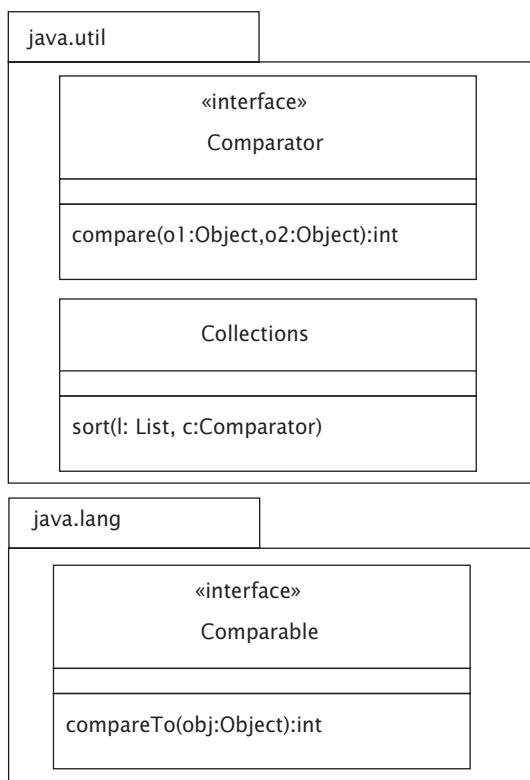
Les classes **Arrays** et **Collections** fournissent une méthode `sort()` statique qui reçoit comme argument un tableau à trier ainsi qu'un **Comparator** optionnel. La méthode `sort()` de la classe **ArrayList** est une méthode d'instance qui détermine le récepteur du message `sort()`. A un autre égard, ces méthodes partagent une stratégie commune qui dépend des interfaces **Comparable** et **Comparator**, comme illustré Figure 21.1.

Les méthodes `sort()` des classes **Arrays** et **Collections** vous permettent de fournir une instance de l'interface **Comparator** si vous le souhaitez. Si vous employez une méthode `sort()` sans fournir une telle instance, elle s'appuiera sur la méthode `compareTo()` de l'interface **Comparable**. Une exception surviendra si vous tentez de trier des éléments sans fournir une instance de **Comparator** et que ces éléments n'implémentent pas l'interface **Comparable**. Mais notez que les types les plus rudimentaires, tels que **String**, implémentent **Comparable**.

Les méthodes `sort()` représentent un exemple de **TEMPLATE METHOD**. Les bibliothèques de classes incluent un algorithme qui vous permet de fournir une étape critique : la comparaison de deux éléments. La méthode `compare()` retourne un nombre inférieur, égal, ou supérieur à 0. Ces valeurs correspondent à l'idée que, dans le sens que vous définissez, l'objet `o1` est inférieur, égal, ou supérieur à l'objet `o2`. Par exemple, le code suivant trie une collection de fusées en fonction de leur apogée

Figure 21.1

La méthode `sort()` de la classe `Collections` utilise les interfaces présentées ici.



puis de leur nom (le constructeur de `Rocket` reçoit le nom, la masse, le prix, l'apogée et la poussée de la fusée) :

```

package app.templateMethod;

import java.util.Arrays;
import com.oozinoz.firework.Rocket;
import com.oozinoz.utility.Dollars;

public class ShowComparator {
    public static void main(String args[]) {
        Rocket r1 = new Rocket(
            "Sock-it", 0.8, new Dollars(11.95), 320, 25);
        Rocket r2 = new Rocket(
            "Sprocket", 1.5, new Dollars(22.95), 270, 40);
        Rocket r3 = new Rocket(
            "Mach-it", 1.1, new Dollars(22.95), 1000, 70);
    }
}

```

```
Rocket r4 = new Rocket(
    "Pocket", 0.3, new Dollars(4.95), 150, 20);
Rocket[] rockets = new Rocket[] { r1, r2, r3, r4 };

System.out.println("Triées par apogée : ");
Arrays.sort(rockets, new ApogeeComparator());
for (int i = 0; i < rockets.length; i++)
    System.out.println(rockets[i]);
System.out.println();
System.out.println("Triées par nom : ");
Arrays.sort(rockets, new NameComparator());
for (int i = 0; i < rockets.length; i++)
    System.out.println(rockets[i]);
}
```

Voici le comparateur ApogeeComparator :

```
package app.templateMethod;

import java.util.Comparator;
import com.oozinoz.firework.Rocket;

public class ApogeeComparator implements Comparator {
    // Exercice !
}
```

Voici le comparateur NameComparator :

```
package app.templateMethod;

import java.util.Comparator;
import com.oozinoz.firework.Rocket;

public class NameComparator implements Comparator {
    // Exercice !
}
```

L'affichage du programme dépend de la façon dont Rocket implémente `toString()` mais montre les fusées triées des deux manières :

```
Triées par apogée :
Pocket
Sprocket
Sock-it
Mach-it
```

Triées par nom :

Mach-it
Pocket
Sock-it
Sprocket

Exercice 21.1

Ecrivez le code manquant dans les classes ApogeeComparator et NameComparator pour que le programme puisse trier correctement une collection de fusées.

- *Les solutions des exercices de ce chapitre sont données dans l'Annexe B.*

Le tri est un algorithme général qui, à l'exception d'une étape, n'a rien à voir avec les spécificités de votre domaine ou application. Cette étape critique est la comparaison d'éléments. Aucun algorithme n'inclut, par exemple, d'étape pour comparer les apogées de deux fusées. Votre application doit donc fournir cette étape. Les méthodes `sort()` et l'interface `Comparator` vous permettent d'insérer une étape spécifique dans un algorithme de tri général.

TEMPLATE METHOD ne se limite pas aux cas où seule l'étape manquante est propre à un domaine. Parfois, l'algorithme entier s'applique à un domaine particulier.

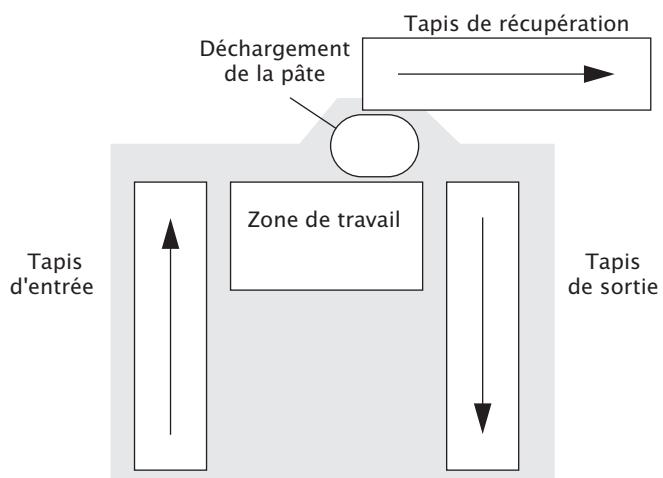
CompléTION D'UN ALGORITHME

Les patterns TEMPLATE METHOD et ADAPTER sont semblables en ce qu'ils permettent tous deux à un développeur de simplifier et de spécifier la façon dont le code d'un autre développeur complète une conception. Dans ADAPTER, un développeur peut spécifier l'interface d'un objet requis par la conception, et un autre peut créer un objet qui fournit l'interface attendue mais en utilisant les services d'une classe existante possédant une interface différente. Dans TEMPLATE METHOD, un développeur peut fournir un algorithme général, et un autre fournir une étape essentielle de l'algorithme. Considérez la presse à étoiles de la Figure 21.2.

La presse à étoiles fabriquée par la société Aster Corporation accepte des moules en métal vides et presse dedans des étoiles de feu d'artifice. La machine possède des trémies qui dispensent les produits chimiques qu'elle mélange en une pâte et presse dans les moules. Lorsqu'elle s'arrête, elle interrompt son traitement du moule qui

Figure 21.2

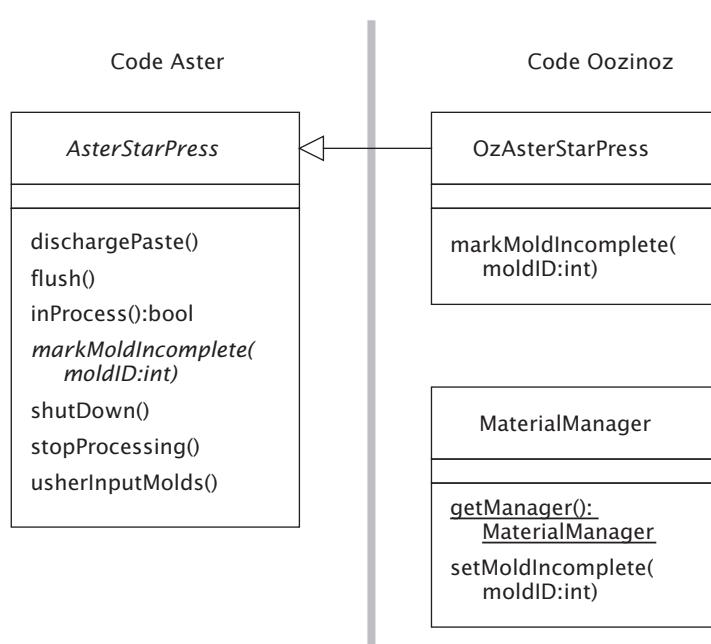
Une presse à étoiles Aster possède des tapis d'entrée et de sortie qui transportent les moules. Oozinoz ajoute un tapis de récupération qui collecte la pâte déchargée.



se trouve dans la zone de travail, et transfère tous les moules de son tapis d'entrée vers son tapis de sortie sans les traiter. Elle décharge ensuite son stock de pâte et rince à grande eau sa zone de travail. Elle orchestre toute cette activité en utilisant un ordinateur de bord et la classe **AsterStarPress** illustrée Figure 21.3.

Figure 21.3

Les presses à étoiles Aster utilisent une classe abstraite dont vous devez dériver une sous-classe pour pouvoir vous en servir chez Oozinoz.



La presse à étoiles Aster est intelligente et indépendante, et a été conçue pour pouvoir opérer dans une unité de production intelligente avec laquelle elle doit communiquer. Par exemple, la méthode `shutDown()` notifie l'unité de production lorsque le moule en cours de traitement est incomplet :

```
public void shutdown() {  
    if (inProcess()) {  
        stopProcessing();  
        markMoldIncomplete(currentMoldID);  
    }  
    usherInputMolds();  
    dischargePaste();  
    flush();  
}
```

La méthode `markMoldIncomplete()` et la classe `AsterStarPress` sont abstraites. Chez Oozinoz, vous devez créer une sous-classe qui implémente la méthode requise et charger ce code dans l'ordinateur de la presse. Vous pouvez implémenter `markMoldIncomplete()` en passant les informations concernant le moule incomplet au singleton `MaterialManager` qui garde trace de l'état matériel.

Exercice 21.2

Ecrivez le code de la méthode `markMoldIncomplete()` de la classe `OzAsterStarPress` :

```
public class OzAsterStarPress extends AsterStarPress {  
    public void markMoldIncomplete(int id) {  
        // Exercice !  
    }  
}
```

Les concepteurs de la presse à étoiles Aster Star connaissent parfaitement le fonctionnement des unités de production pyrotechniques et ont implémenté la communication avec l'unité aux points de traitement appropriés. Il se peut néanmoins que vous ayez besoin d'établir la communication à un point que ces développeurs ont omis.

Hooks

Un **hook** est un appel de méthode placé par un développeur à un point spécifique d'une procédure pour permettre à d'autres développeurs d'y insérer du code. Lorsque vous adaptez le code d'un autre développeur et avez besoin de disposer d'un contrôle à un certain point auquel vous n'avez pas accès, vous pouvez demander un hook. Un développeur serviable insérera un appel de méthode au niveau de ce point et fournira aussi généralement une version stub de la méthode hook pour éviter à d'autres clients de devoir la remplacer.

Considérez la presse Aster qui décharge sa pâte chimique et rince abondamment sa zone de travail lorsqu'elle s'arrête. Elle doit décharger la pâte pour empêcher que celle-ci ne sèche et bloque la machine. Chez Oozinoz, vous récupérez la pâte et la découpez en dés qui serviront de petites étoiles dans des chandelles romaines (une **chandelle romaine** est un tube stationnaire qui contient un mélange de charges explosives et d'étoiles). Une fois la pâte déchargée, vous faites en sorte qu'un robot la place sur un tapis séparé, comme illustré Figure 21.2. Il importe de procéder au déchargement avant que la machine ne lave sa zone de travail. Le problème est que vous voulez prendre le contrôle entre les deux instructions de la méthode `shutdown()` :

```
dischargePaste();
flush();
```

Vous pourriez remplacer `dischargePaste()` par une méthode qui ajoute un appel pour collecter la pâte :

```
public void dischargePaste() {
    super.dischargePaste();
    getFactory().collectPaste();
}
```

Cette méthode insère une étape après le déchargement de la pâte. Cette étape utilise un singleton Factory pour collecter la pâte. Lorsque la méthode `shutdown()` s'exécutera, le robot recueillera la pâte déchargée avant que la presse ne soit rincée. Malheureusement, le code de `dischargePaste()` introduit un risque. Les développeurs de chez Aster ne sauront certainement pas que vous avez défini ainsi cette méthode. S'ils modifient leur code pour décharger la pâte à un moment où vous ne voulez pas la collecter, une erreur surviendra.

Les développeurs cherchent généralement à résoudre les problèmes en écrivant du code. Mais ici, il s'agit de résoudre un problème potentiel en communiquant les uns avec les autres.

Exercice 21.3

Rédigez une note à l'attention des développeurs de chez Aster leur demandant d'introduire un changement qui vous permettra de collecter la pâte déchargée en toute sécurité avant que la machine ne rince sa zone de travail.

L'étape fournie par une sous-classe dans TEMPLATE METHOD peut être nécessaire pour compléter l'algorithme ou peut représenter une étape optionnelle qui s'insère dans le code d'une sous-classe, souvent à la demande d'un autre développeur. Bien que l'objectif de ce pattern soit de laisser une classe séparée définir une partie d'un algorithme, vous pouvez aussi l'appliquer lorsque vous refactorisez un algorithme apparaissant dans plusieurs méthodes.

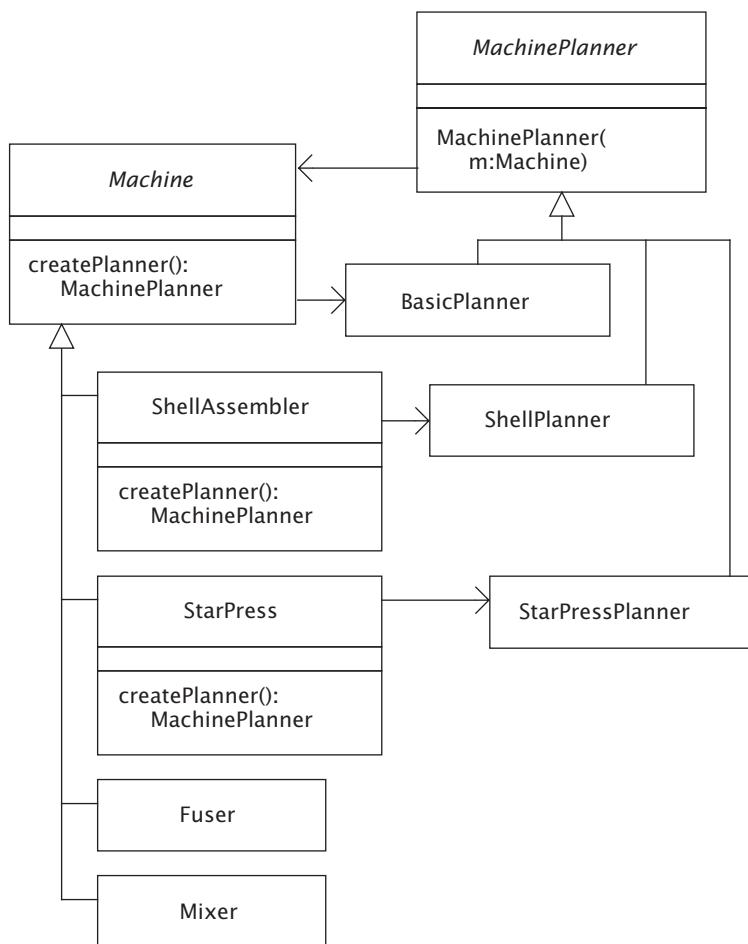
Refactorisation pour appliquer TEMPLATE METHOD

Lorsque TEMPLATE METHOD est appliqué, vous trouverez des hiérarchies de classes où une super-classe fournit la structure générale d'un algorithme et où des sous-classes en fournissent certaines étapes. Vous pouvez adopter cette approche, refactorisant en vue d'appliquer TEMPLATE METHOD, lorsque vous trouvez des algorithmes similaires dans des méthodes différentes (*refactoriser* consiste à transformer des programmes en des programmes équivalents mais mieux conçus). Considérez les hiérarchies parallèles Machine et MachinePlanner introduites au Chapitre 16, consacré au pattern FACTORY METHOD. Comme le montre la Figure 21.4, la classe Machine fournit une méthode `createPlanner()` en tant que FACTORY METHOD qui retourne une sous-classe appropriée de MachinePlanner.

Deux des sous-classes de Machine instancient des sous-classes spécifiques de la hiérarchie MachinePlanner lorsqu'il leur est demandé de créer un planificateur. Ces classes, `ShellAssembler` et `StarPress`, posent un même problème en ce qu'elles ne peuvent créer un `MachinePlanner` qu'à la demande.

Figure 21.4

Un objet *Machine* peut créer une instance appropriée de *MachinePlanner* pour lui-même.



Si vous examinez le code de ces classes, vous noterez que les sous-classes emploient des techniques similaires pour procéder à une initialisation paresseuse (*lazy-initialization*) d'un planificateur. Par exemple, la classe *ShellAssembler* possède une méthode *getPlanner()* qui initialise un membre *planner* :

```

public ShellPlanner getPlanner() {
    if (planner == null)
        planner = new ShellPlanner(this);
    return planner;
}
  
```

Dans la classe `ShellPlanner`, `planner` est de type `ShellPlanner`. La classe `StarPress` comprend aussi un membre `planner` mais le déclare comme étant de type `StarPressPlanner`. La méthode `getPlanner()` de la classe `StarPress` opère aussi une initialisation paresseuse de l'attribut `planner` :

```
public StarPressPlanner getPlanner() {  
    if (planner == null)  
        planner = new StarPressPlanner(this);  
    return planner;  
}
```

Les autres sous-classes de `Machine` adoptent une approche analogue pour créer un planificateur seulement lorsqu'il est nécessaire. Cela présente une opportunité de refactorisation, vous permettant de nettoyer et de réduire votre code. En supposant que vous décidiez d'ajouter à la classe `Machine` un attribut `planner` de type `MachinePlanner`, cela vous permettrait de supprimer cet attribut des sous-classes et d'éliminer les méthodes `getPlanner()` existantes.

Exercice 21.4

Ecrivez le code de la méthode `getPlanner()` de la classe `Machine`.

Vous pouvez souvent refactoriser votre code en une instance de `TEMPLATE METHOD` en rendant abstraite la structure générale de méthodes qui se ressemblent, c'est-à-dire en plaçant cette structure dans une super-classe et en laissant aux sous-classes le soin de fournir l'étape qui diffère dans leur implémentation de l'algorithme.

Résumé

L'objectif de `TEMPLATE METHOD` est de définir un algorithme dans une méthode, laissant certaines étapes abstraites, non définies, ou définies dans une interface, de sorte que d'autres classes puissent se charger de les implémenter.

Ce pattern fonctionne comme un contrat entre les développeurs. Un développeur fournit la structure générale d'un algorithme, et un autre en fournit une certaine étape. Il peut s'agir d'une étape qui complète l'algorithme ou qui sert de hook vous permettant d'insérer votre code à des points spécifiques de la procédure.

TEMPLATE METHOD n’implique pas que vous deviez écrire la méthode modèle avant les sous-classes d’implémentation. Il peut arriver que vous tombiez sur des méthodes similaires dans une hiérarchie existante. Vous pourriez alors en extraire la structure générale d’un algorithme et la placer dans une super-classe, appliquant ce pattern pour simplifier et réorganiser votre code.

STATE

L'état d'un objet est une combinaison des valeurs courantes de ses attributs. Lorsque vous appelez une méthode `set...` d'un objet ou assignez une valeur à l'un de ses champs, vous changez son état. Les objets modifient souvent aussi eux-mêmes leur état lorsque leurs méthodes s'exécutent.

Le terme *état (state)* est parfois employé pour désigner un attribut changeant d'un objet. Par exemple, nous pourrions dire qu'une machine est dans un état actif ou inactif. Dans un tel cas, la partie changeante de l'état de l'objet est l'aspect le plus important de son comportement. En conséquence, la logique qui dépend de l'état de l'objet peut se trouver répartie dans de nombreuses méthodes de la classe. Une logique semblable ou identique peut ainsi apparaître de nombreuses fois, augmentant le travail de maintenance du code.

Une façon d'éviter cet éparpillement de la logique dépendant de l'état d'un objet est d'introduire un nouveau groupe de classes, chacune représentant un état différent. Il faut ensuite placer le comportement spécifique à un état dans la classe appropriée.

L'objectif du pattern STATE est de distribuer la logique dépendant de l'état d'un objet à travers plusieurs classes qui représentent chacune un état différent.

Modélisation d'états

Lorsque vous modéliez un objet dont l'état est important, il se peut qu'il dispose d'une variable qui garde trace de la façon dont il devrait se comporter, selon son état. Cette variable apparaît peut-être dans des instructions `if` en cascade complexes qui se concentrent sur la façon de réagir aux événements expérimentés par l'objet.

Cette approche de modélisation de l'état présente deux problèmes. Premièrement, les instructions `if` peuvent devenir complexes ; deuxièmement, lorsque vous ajustez la façon dont vous modélez l'état, il est souvent nécessaire d'ajuster les instructions `if` de plusieurs méthodes. Le pattern STATE offre une approche plus claire et plus simple, utilisant une opération distribuée. Il vous permet de modéliser des états en tant qu'objets, encapsulant la logique dépendant de l'état dans des classes distinctes. Avant de voir ce pattern à l'œuvre, il peut être utile d'examiner un système qui modélise des états sans y recourir. Dans la section suivante, nous refactoriserons ce code pour déterminer si STATE peut améliorer la conception.

Considérez le logiciel d'Oozinoz qui modélise les états d'une porte de carrousel. Un **carrousel** est un grand rack intelligent qui accepte des produits par une porte et les stocke d'après leur code-barres. La porte fonctionne au moyen d'un seul bouton. Lorsqu'elle est fermée, toucher le bouton provoque son ouverture. Le toucher avant qu'elle soit complètement ouverte la fait se fermer. Lorsqu'elle est complètement ouverte, elle commence à se fermer automatiquement après deux secondes. Vous pouvez empêcher cela en touchant le bouton pendant que la porte est encore ouverte. La Figure 22.1 montre les états et les transitions de cette porte. Le code correspondant est présenté un peu plus loin.

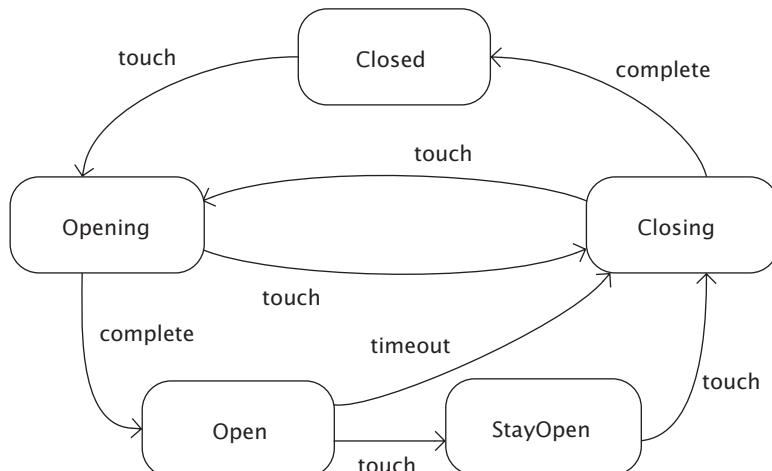


Figure 22.1

La porte du carrousel dispose d'un bouton de contrôle réagissant au toucher et permettant de changer ses états.

Ce diagramme est une *machine à états* UML. De tels diagrammes peuvent être beaucoup plus informatifs qu'une description textuelle.

Exercice 22.1

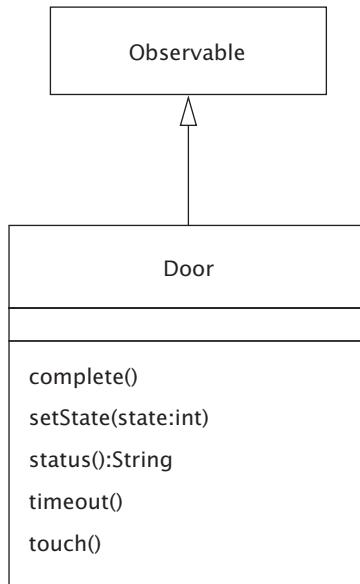
Supposez que vous ouvriez la porte et passiez une caisse de produits de l'autre côté. Y a-t-il un moyen de faire en sorte que la porte commence à se fermer avant le délai de deux secondes ?

- *Les solutions des exercices de ce chapitre sont données dans l'Annexe B.*

Vous pourriez introduire dans le logiciel du carrousel un objet Door qu'il actualiserait avec les changements d'état de la porte. La Figure 22.2 présente la classe Door.

Figure 22.2

La classe Door modélise une porte de carrousel, s'appuyant sur les événements de changement d'état envoyés par le carrousel.



La classe Door est Observable de sorte que les clients, telle une interface GUI, puissent afficher l'état de la porte. La définition de cette classe établit les différents états que peut prendre la porte :

```
package com.oozinoz.carousel;
import java.util.Observable;
```

```
public class Door extends Observable {  
    public final int CLOSED = -1;  
    public final int OPENING = -2;  
    public final int OPEN = -3;  
    public final int CLOSING = -4;  
    public final int STAYOPEN = -5;  
  
    private int state = CLOSED;  
  
    // ...  
}
```

(Vous pourriez choisir d'utiliser un type énuméré si vous programmez avec Java 5.) La description textuelle de l'état de la porte dépendra évidemment de l'état dans lequel elle se trouve :

```
public String status() {  
    switch (state) {  
        case OPENING:  
            return "Opening";  
        case OPEN:  
            return "Open";  
        case CLOSING:  
            return "Closing";  
        case STAYOPEN:  
            return "StayOpen";  
        default:  
            return "Closed";  
    }  
}
```

Lorsqu'un utilisateur touche le bouton du carrousel, ce dernier génère un appel de la méthode `touch()` d'un objet `Door`. Le code de `Door` pour une transition d'état reflète les informations de la Figure 22.1 :

```
public void touch() {  
    switch (state) {  
        case OPENING:  
        case STAYOPEN:  
            setState(CLOSING);  
            break;  
        case CLOSING:  
        case CLOSED:  
            setState(OPENING);  
            break;  
    }  
}
```

```
        case OPEN:
            setState(STAYOPEN);
            break;
        default:
            throw new Error("Ne peut se produire");
    }
}
```

La méthode `setState()` de la classe `Door` notifie aux observateurs le changement d'état de la porte :

```
private void setState(int state) {
    this.state = state;
    setChanged();
    notifyObservers();
}
```

Exercice 22.2

Ecrivez le code des méthodes `complete()` et `timeout()` de la classe `Door`.

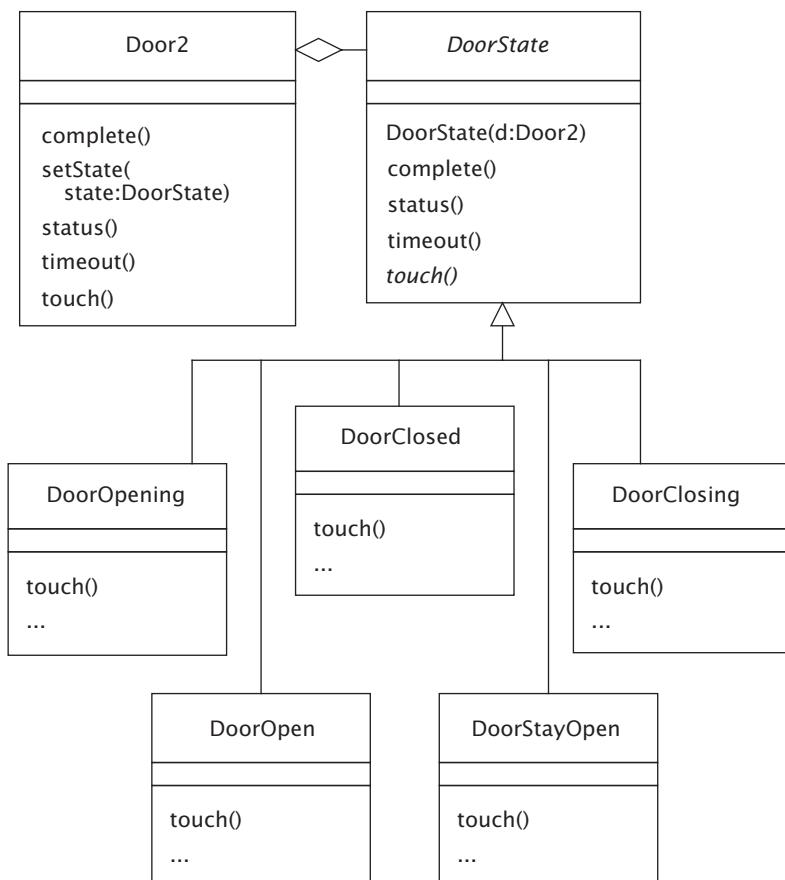
Refactorisation pour appliquer STATE

Le code de `Door` est quelque peu complexe car l'utilisation de la variable `state` est répartie à travers toute la classe. En outre, il peut être difficile de comparer les méthodes de transition d'état, plus particulièrement `touch()`, avec la machine à états de la Figure 22.1. Le pattern STATE peut vous aider à simplifier ce code. Pour l'appliquer dans cet exemple, il faut définir chaque état de la porte en tant que classe distincte, comme illustré Figure 22.3.

La refactorisation illustrée dans cette figure crée une classe séparée pour chaque état dans lequel la porte peut se trouver. Chacune d'elles contient la logique permettant de répondre au toucher du bouton pendant que la porte est dans un certain état. Par exemple, le fichier `DoorClosed.java` contient le code suivant :

```
package com.oozinoz.carousel;
public class DoorClosed extends DoorState {
    public DoorClosed(Door2 door) {
        super(door);
    }

    public void touch() {
        door.setState(door.OPENING);
    }
}
```

**Figure 22.3**

Le diagramme représente les états de la porte en tant que classes dans une structure qui reflète la machine à états de la porte.

La méthode `touch()` de la classe `DoorClosed` informe l'objet `Door2` du nouvel état de la porte. Cet objet est celui reçu par le constructeur de `DoorClosed`. Cette approche implique que chaque objet état contienne une référence à un objet `Door2` pour pouvoir informer la porte des transitions d'état. Un objet état ne peut donc s'appliquer ici qu'à une seule porte. La prochaine section décrit comment modifier cette conception pour qu'un même ensemble d'états suffise pour un nombre quelconque de portes.

La génération d'un objet `Door2` doit s'accompagner de la création d'une suite d'états appartenant à la porte :

```
package com.oozinoz.carousel;
import java.util.Observable;

public class Door2 extends Observable {
    public final DoorState CLOSED = new DoorClosed(this);
    public final DoorState CLOSING = new DoorClosing(this);
    public final DoorState OPEN = new DoorOpen(this);
    public final DoorState OPENING = new DoorOpening(this);
    public final DoorState STAYOPEN = new DoorStayOpen(this);

    private DoorState state = CLOSED;
    // ...
}
```

La classe abstraite `DoorState` requiert des sous-classes pour implémenter `touch()`. Cela est cohérent avec la machine à états, dans laquelle tous les états possèdent une transition `touch()`. Cette classe ne définit pas les autres transitions, les laissant stub, pour permettre aux sous-classes de les remplacer ou d'ignorer les messages non pertinents :

```
package com.oozinoz.carousel;

public abstract class DoorState {
    protected Door2 door;

    public abstract void touch();

    public void complete() { }

    public void timeout() { }

    public String status() {
        String s = getClass().getName();
        return s.substring(s.lastIndexOf('.') + 1);
    }

    public DoorState(Door2 door) {
        this.door = door;
    }
}
```

Notez que la méthode `status()` fonctionne pour tous les états et est beaucoup plus simple qu'avant la refactorisation.

La nouvelle conception ne change pas le rôle d'un objet `Door2` en ce qu'il reçoit toujours les changements d'état de la part du carrousel, mais maintenant il les passe simplement à son objet `state` courant :

```
package com.oozinoz.carousel;
import java.util.Observable;

public class Door2 extends Observable {
    // variables et constructeur...

    public void touch() {
        state.touch();
    }

    public void complete() {
        state.complete();
    }

    public void timeout() {
        state.timeout();
    }

    public String status() {
        return state.status();
    }

    protected void setState(DoorState state) {
        this.state = state;
        setChanged();
        notifyObservers();
    }
}
```

Les méthodes `touch()`, `complete()`, `timeout()` et `status()` illustrent le rôle du polymorphisme dans cette conception. Chacune d'elles est toujours un genre d'instruction `switch`. Dans chaque cas, l'opération est fixe, mais la classe du récepteur, c'est-à-dire la classe de `state`, varie quant à elle. La règle du polymorphisme est que la méthode qui s'exécute dépend de la signature de l'opération et de la classe du récepteur. Que se passe-t-il lorsque vous appelez `touch()` ? Cela dépend de l'état de la porte. Le code accomplit toujours un "switch", mais, en s'appuyant sur le polymorphisme, il est plus simple qu'auparavant.

La méthode `setState()` de la classe `Door2` est maintenant utilisée par des sous-classes de `DoorState`. Celles-ci ressemblent à leurs contreparties dans la machine à états de la Figure 22.1. Par exemple, le code de `DoorOpen` gère les appels de `touch()` et `timeout()`, les deux transitions de l'état Open dans la machine :

```
package com.oozinoz.carousel;
public class DoorOpen extends DoorState {
```

```
public DoorOpen(Door2 door) {  
    super(door);  
}  
  
public void touch() {  
    door.setState(door.STAYOPEN);  
}  
  
public void timeout() {  
    door.setState(door.CLOSING);  
}  
}
```

Exercice 22.3

Ecrivez le code de `DoorClosing.java`.

La nouvelle conception donne lieu à un code beaucoup plus simple, mais il se peut que vous ne soyez pas complètement satisfait du fait que les "constantes" utilisées par la classe `Door` soient en fait des variables locales.

Etats constants

Le pattern STATE répartit la logique dépendant de l'état d'un objet dans plusieurs classes qui représentent les différents états de l'objet. Il ne spécifie toutefois pas comment gérer la communication et les dépendances entre les objets état et l'objet central auquel ils s'appliquent. Dans la conception précédente, chaque classe d'état acceptait un objet `Door` dans son constructeur. Les objets état conservaient cet objet et s'en servaient pour actualiser l'état de la porte. Cette approche n'est pas forcément mauvaise, mais elle a pour effet qu'instancier un objet `Door` entraîne l'instantiation d'un ensemble complet d'objets `DoorState`. Vous pourriez préférer une conception qui crée un seul ensemble statique d'objets `DoorState` et requière que la classe `Door` gère toutes les actualisations résultant des changements d'état.

Une façon de rendre les objets état constants est de faire en sorte que les classes d'état identifient simplement l'état suivant, laissant le soin à la classe `Door` d'actualiser sa variable `state`. Dans une telle conception, la méthode `touch()` de la classe `Door`, par exemple, actualise la variable `state` comme suit :

```
public void touch() {  
    state = state.touch();  
}
```

Notez que le type de retour de la méthode `touch()` de la classe `Door` est `void`. Les sous-classes de `DoorState` implémenteront aussi `touch()` mais retourneront une valeur `DoorState`. Par exemple, voici à présent le code de la méthode `touch()` de `DoorOpen` :

```
public DoorState touch() {  
    return DoorState.STAYOPEN;  
}
```

Dans cette conception, les objets `DoorState` ne conservent pas de référence vers un objet `Door`, aussi l'application requiert-elle une seule instance de chaque objet `DoorState`.

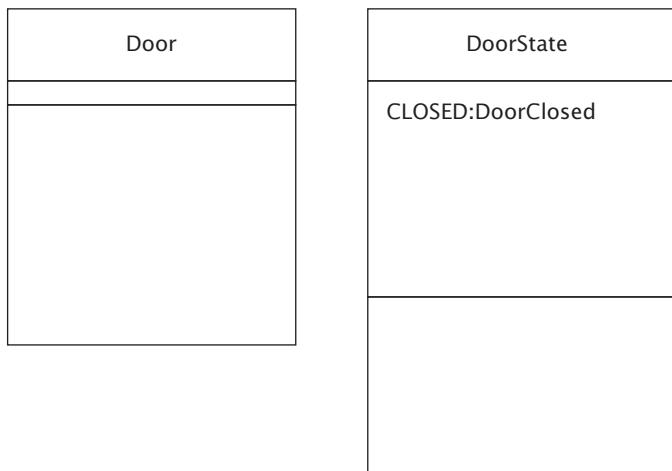
Une autre approche pour rendre les objets `DoorState` constants est de faire passer l'objet `Door` central pendant les transitions d'état. Pour cela, il faut ajouter un paramètre `Door` aux méthodes `complete()`, `timeout()` et `touch()`. Elles recevront alors l'objet `Door` en tant que paramètre et actualiseront son état sans conserver de référence vers lui.

Exercice 22.4

Complétez le diagramme de classes de la Figure 22.4 pour représenter une conception où les objets `DoorState` sont constants et qui fait passer un objet `Door` pendant les transitions d'état.

Figure 22.4

Une fois complété, ce diagramme représentera une conception qui rend les états de la porte constants.



Lorsque vous appliquez le pattern STATE, vous disposez d'une liberté totale dans la façon dont votre conception organise la communication des changements d'état. Les classes d'état peuvent conserver une référence à l'objet central dont l'état est modélisé. Sinon, vous pouvez faire passer cet objet durant les transitions. Vous pouvez aussi faire en sorte que les sous-classes soient de simples fournisseurs d'informations déterminant l'état suivant mais n'actualisant pas l'objet central. L'approche que vous choisissez dépend du contexte de votre application ou de considérations esthétiques.

Si vos états sont utilisés par différents threads, assurez-vous que vos méthodes de transition sont synchronisées pour garantir l'absence de conflit lorsque deux threads tentent de modifier l'état au même moment.

La puissance du pattern STATE est de permettre la centralisation de la logique de différents états dans une même classe.

Résumé

De manière générale, l'état d'un objet dépend de la valeur collective de ses variables d'instance. Dans certains cas, la plupart des attributs de l'objet sont assez statiques une fois définis, à l'exception d'un attribut qui est dynamique et joue un rôle prédominant dans la logique de la classe. Cet attribut peut représenter l'état de l'objet tout entier et peut même être nommé `state`.

Une variable d'état dominante peut exister lorsqu'un objet modélise une entité du monde réel dont l'état est important, telle qu'une transaction ou une machine. La logique qui dépend de l'état de l'objet peut alors apparaître dans de nombreuses méthodes. Vous pouvez simplifier un tel code en plaçant les comportements spécifiques aux différents états dans une hiérarchie d'objets état. Chaque classe d'état peut ainsi contenir le comportement pour un seul état du domaine. Cela permet également d'établir une correspondance directe entre les classes d'état et les états d'une machine à états.

Pour gérer les transitions entre les états, vous pouvez laisser l'objet central conserver des références vers un ensemble d'états. Ou bien, dans les méthodes de transition d'état, vous pouvez faire passer l'objet central dont l'état change. Vous pouvez sinon faire des classes d'état des fournisseurs d'informations qui indiquent simplement un état subséquent sans actualiser l'objet central. Quelle que soit la manière dont vous procédez, le pattern STATE simplifie votre code en distribuant une opération à travers une collection de classes qui représentent les différents états d'un objet.

STRATEGY

Une stratégie est un plan, ou une approche, pour atteindre un but en fonction de certaines conditions initiales. Elle est donc semblable à un algorithme, lequel est une procédure générant un résultat à partir d'un ensemble d'entrées. Habituellement, une stratégie dispose d'une plus grande latitude qu'un algorithme pour accomplir son objectif. Cette latitude signifie également que les stratégies apparaissent souvent par groupes, ou familles, d'alternatives.

Lorsque plusieurs stratégies apparaissent dans un programme, le code peut devenir complexe. La logique qui entoure les stratégies doit sélectionner l'une d'elles, et ce code de sélection peut lui-même devenir complexe. L'exécution de plusieurs stratégies peut emprunter différents chemins dans le code, lequel peut même être contenu dans une seule méthode. Lorsque la sélection et l'exécution de diverses méthodes conduisent à un code complexe, vous pouvez appliquer le pattern STRATEGY pour le simplifier.

L'opération stratégique définit les entrées et les sorties d'une stratégie mais laisse l'implémentation aux classes individuelles. Les classes qui implémentent les diverses approches implémentent la même opération et sont donc interchangeables, présentant des stratégies différentes mais la même interface aux clients. Le pattern STRATEGY permet à une famille de stratégies de coexister sans que leurs codes respectifs s'entremêlent. Il sépare aussi la logique de sélection d'une stratégie des stratégies elles-mêmes.

L'objectif du pattern STRATEGY est d'encapsuler des approches, ou stratégies, alternatives dans des classes distinctes qui implémentent chacune une opération commune.

Modélisation de stratégies

Le pattern STRATEGY aide à organiser et à simplifier le code en encapsulant différentes approches d'un problème dans plusieurs classes. Avant de le voir à l'œuvre, il peut être utile d'examiner un programme qui modélise des stratégies sans l'appliquer. Dans la section suivante, nous refactoriserons ce code en appliquant STRATEGY pour améliorer sa qualité.

Considérez la politique publicitaire d'Oozinoz qui suggère aux clients qui visitent son site Web ou prennent contact avec son centre d'appels quel artifice acheter. Oozinoz utilise deux moteurs de recommandation du commerce pour déterminer l'article approprié à proposer. La classe `Customer` choisit et applique un des deux moteurs.

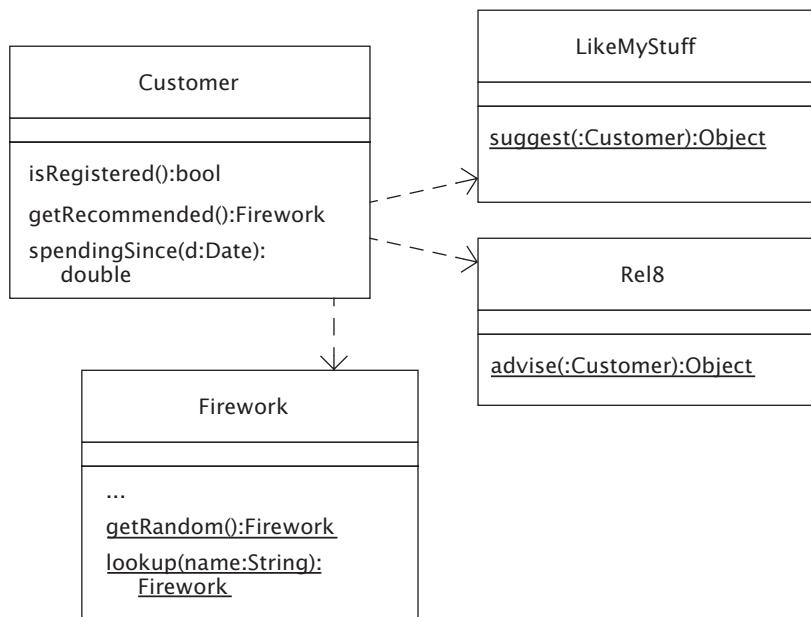


Figure 23.1

La classe Customer s'appuie sur d'autres classes pour ses recommandations, parmi lesquelles deux moteurs de recommandation du commerce.

Un des moteurs de recommandation, `Rel8`, suggère un achat en se fondant sur les similitudes du client avec d'autres clients. Pour que cela fonctionne, le client doit s'être enregistré au préalable et avoir fourni des informations sur ses préférences en matière d'artifices et autres distractions. S'il n'est pas encore enregistré, Oozinoz

utilise l'autre moteur, `LikeMyStuff`, qui suggère un achat sur la base des achats récents du client. Si aucun des deux moteurs ne dispose de suffisamment de données pour assurer sa fonction, le logiciel de publicité choisit un artifice au hasard. Une promotion spéciale peut néanmoins avoir la priorité sur toutes ces considérations, mettant en avant un artifice particulier qu'Oozinoz cherche à vendre. La Figure 23.1 présente les classes qui collaborent pour suggérer un artifice à un client.

Les moteurs `LikeMyStuff` et `Rel8` acceptent un objet `Customer` et suggèrent quel artifice proposer au client. Tous deux sont configurés chez Oozinoz pour gérer des artifices, mais `LikeMyStuff` requiert une base de données tandis que `Rel8` travaille essentiellement à partir d'un modèle objet. Le code de la méthode `getRecommended()` de la classe `Customer` reflète la politique publicitaire d'Oozinoz :

```
public Firework getRecommended() {
    // En cas de promotion d'un artifice particulier, le retourner.
    try {
        Properties p = new Properties();
        p.load(ClassLoader.getSystemResourceAsStream(
            "config/strategy.dat"));
        String promotedName = p.getProperty("promote");

        if (promotedName != null) {
            Firework f = Firework.lookup(promotedName);
            if (f != null) return f;
        }
    } catch (Exception ignored) {
        // Si la ressource est manquante ou n'a pas été chargée,
        // se rabattre sur l'approche suivante.
    }

    // Si le client est enregistré, le comparer aux autres clients.
    if (isRegistered()) {
        return (Firework) Rel8.advise(this);
    }

    // Vérifier les achats du client sur l'année écoulée.
    Calendar cal = Calendar.getInstance();
    cal.add(Calendar.YEAR, -1);
    if (spendingSince(cal.getTime()) > 1000)
        return (Firework) LikeMyStuff.suggest(this);

    // Retourner n'importe quel artifice.
    return Firework.getRandom();
}
```

Ce code est extrait du package com.oozinoz.recommendation de la base de code Oozinoz accessible à l'adresse **www.oozinoz.com**. La méthode `getRecommended()` s'attend à ce que, s'il y a une promotion, elle soit nommée dans un fichier `strategy.dat` dans un répertoire `config`. Voici à quoi ressemblerait un tel fichier :

```
promote=JSquirrel
```

En l'absence de ce fichier, la méthode utilise le moteur `Re18` si le client est inscrit. Si le client n'est pas inscrit, elle utilise le moteur `LikeMyStuff` lorsque le client a déjà dépensé un certain montant au cours de l'année passée. Si aucune meilleure recommandation n'est possible, le code sélectionne et propose un artifice quelconque. Cette méthode fonctionne, et vous avez probablement déjà vu pire comme code, mais nous pouvons certainement l'améliorer.

Refactorisation pour appliquer STRATEGY

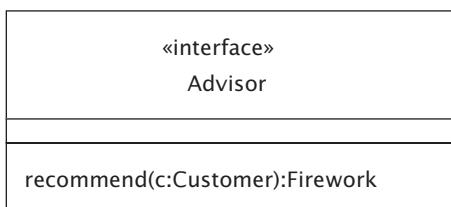
La méthode `getRecommended()` présente plusieurs problèmes. D'abord, elle est longue, au point que des commentaires doivent expliquer ses différentes parties. Les méthodes courtes sont plus faciles à comprendre et ont rarement besoin d'être expliquées, elles sont donc généralement préférables aux méthodes longues. Ensuite, non seulement elle choisit une stratégie mais elle l'exécute également, ce qui constitue deux fonctions distinctes qui peuvent être séparées. Vous pouvez simplifier ce code en appliquant **STRATEGY**. Pour cela, vous devez :

- créer une interface qui définit l'opération stratégique ;
- implémenter l'interface avec des classes qui représentent chacune une stratégie ;
- refactoriser le code pour sélectionner et utiliser une instance de la classe de stratégie appropriée.

Supposez que vous créez une interface `Advisor`, comme illustré Figure 23.2.

Figure 23.2

L'interface Advisor définit une opération que diverses classes peuvent implémenter avec différentes stratégies.



L'interface Advisor déclare qu'une classe qui l'implémente peut accepter un client et recommander un artifice. L'étape suivante consiste à refactoriser la méthode `getRecommended()` de la classe `Customer` pour créer des classes représentant chacune des stratégies de recommandation. Chaque classe fournit une implémentation différente de la méthode `recommend()` spécifiée par l'interface Advisor.

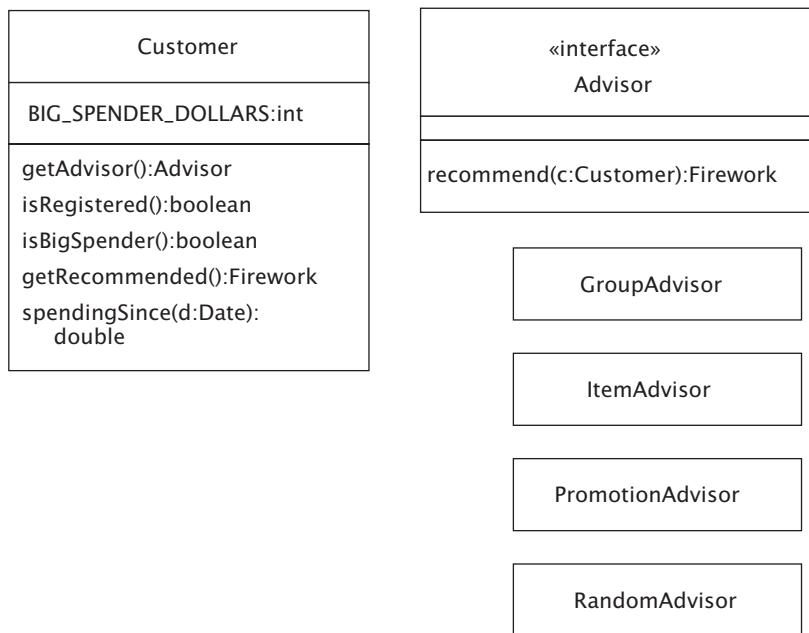


Figure 23.3

Complétez ce diagramme pour montrer la refactorisation du logiciel de recommandation, avec les stratégies apparaissant comme implémentations d'une interface commune.

Une fois que vous disposez des classes de stratégie, vous devez y placer le code de la méthode `getRecommended()` de la classe `Customer`. Les deux classes les plus simples sont `GroupAdvisor` et `ItemAdvisor`. Elles doivent seulement envelopper les appels pour les moteurs de recommandation. Une interface ne pouvant définir que des méthodes d'instance, `GroupAdvisor` et `ItemAdvisor` doivent être instanciées pour supporter l'interface `Advisor`. Comme un seul objet de chaque classe est nécessaire, `Customer` devrait inclure une seule instance statique de chaque classe.

La Figure 23.4 illustre une conception pour ces classes.

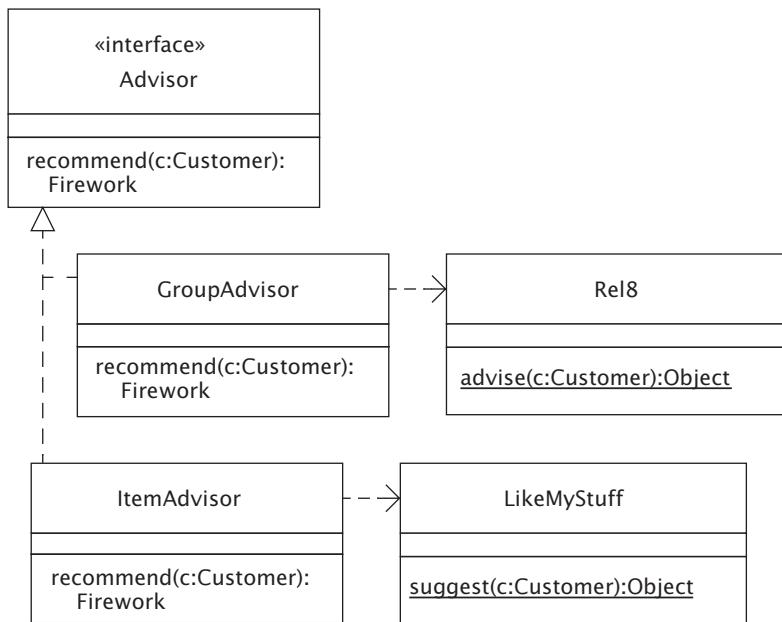


Figure 23.4

Les implémentations de l'interface Advisor fournissent l'opération stratégique recommend(), s'appuyant sur les moteurs de recommandation.

Les classes ...Advisor traduisent les appels de `recommend()` en interfaces requises par les moteurs sous-jacents. Par exemple, la classe **GroupAdvisor** traduit ces appels en l'interface `advise()` requise par le moteur **Rel8** :

```

public Firework recommend(Customer c) {
    return (Firework) Rel8.advise(c);
}
  
```

Exercice 23.2

Outre le pattern STRATEGY, quel autre pattern apparaît dans les classes **GroupAdvisor** et **ItemAdvisor** ?

Les classes `GroupAdvisor` et `ItemAdvisor` opèrent en traduisant un appel de la méthode `recommend()` en un appel d'un moteur de recommandation. Il faut aussi créer une classe `PromotionAdvisor` et une classe `RandomAdvisor`, en refactorisant le code de la méthode `getRecommended()` de `Customer`. A l'instar de `GroupAdvisor` et `ItemAdvisor`, ces classes fournissent aussi l'opération `recommend()`.

Le constructeur de `PromotionAdvisor` devrait déterminer s'il existe une promotion en cours. Vous pourriez ensuite ajouter à cette classe une méthode `hasItem()` indiquant s'il y a un article en promotion :

```
public class PromotionAdvisor implements Advisor {  
    private Firework promoted;  
  
    public PromotionAdvisor() {  
        try {  
            Properties p = new Properties();  
            p.load(ClassLoader.getSystemResourceAsStream(  
                "config/strategy.dat"));  
            String promotedFireworkName = p.getProperty("promote");  
            if (promotedFireworkName != null)  
                promoted = Firework.lookup(promotedFireworkName);  
        } catch (Exception ignored) {  
            // Ressource introuvable ou non chargée  
            promoted = null;  
        }  
    }  
  
    public boolean hasItem() {  
        return promoted != null;  
    }  
  
    public Firework recommend(Customer c) {  
        return promoted;  
    }  
}
```

La classe `RandomAdvisor` est simple :

```
public class RandomAdvisor implements Advisor {  
    public Firework recommend(Customer c) {  
        return Firework.getRandom();  
    }  
}
```

La refactorisation de `Customer` permet de séparer la *sélection* d'une stratégie de son *utilisation*. Un attribut `advisor` d'un objet `Customer` contient le choix courant de la stratégie à appliquer. La classe `Customer2` refactorisée procède à une

initialisation paresseuse de cet attribut avec une logique qui reflète la politique publicitaire d'Oozinoz :

```
private Advisor getAdvisor() {  
    if (advisor == null) {  
        if (promotionAdvisor.hasItem())  
            advisor = promotionAdvisor;  
        else if (isRegistered())  
            advisor = groupAdvisor;  
        else if (isBigSpender())  
            advisor = itemAdvisor;  
        else  
            advisor = randomAdvisor;  
    }  
    return advisor;  
}
```

Exercice 23.3

Ecrivez le nouveau code de la méthode `Customer.getRecommended()`.

Comparaison de **STRATEGY** et **STATE**

Le code refactorisé consiste presque entièrement en des méthodes simples dans des classes simples. Cela représente un avantage en soi et facilite l'ajout de nouvelles stratégies. La refactorisation se fonde principalement sur le principe de distribuer une opération à travers un groupe de classes associées. A cet égard, **STRATEGY** est identique à **STATE**. En fait, certains développeurs se demandent même si ces deux patterns sont vraiment différents.

D'un côté, la différence entre modéliser des états et modéliser des stratégies peut paraître subtile. En effet, **STATE** et **STRATEGY** semblent faire une utilisation du polymorphisme quasiment identique sur le plan structurel.

D'un autre côté, dans le monde réel, les stratégies et les états représentent clairement des concepts différents, et cette différence donne lieu à divers problèmes de modélisation. Par exemple, les transitions sont importantes lorsqu'il s'agit de modéliser des états tandis qu'elles sont hors de propos lorsqu'il s'agit de choisir une stratégie. Une autre différence est que **STRATEGY** peut permettre à un client de sélectionner ou de fournir une stratégie, une idée qui s'applique rarement à **STATE**. Etant donné que ces deux patterns n'ont pas le même objectif, nous continuerons de

les considérer comme différents. Mais vous devez savoir que tout le monde ne reconnaît pas cette distinction.

Comparaison de STRATEGY et TEMPLATE METHOD

Le Chapitre 21, consacré à TEMPLATE METHOD, a pris le triage comme exemple de TEMPLATE METHOD. Vous pouvez utiliser l'algorithme `sort()` de la classe Arrays ou Collection pour trier n'importe quelle liste d'objets, dès lors que vous fournissez une étape pour comparer deux objets. Vous pourriez avancer que lorsque vous fournissez une étape de comparaison pour un algorithme de tri, vous changez la stratégie. En supposant par exemple que vous vendiez des fusées, le fait de les présenter triées par prix ou triées par poussées représente deux stratégies marketing différentes.

Exercice 23.4

Expliquez en quoi la méthode `Arrays.sort()` constitue un exemple de TEMPLATE METHOD et/ou de STRATEGY.

Résumé

Il arrive que la logique qui modélise des stratégies alternatives apparaisse dans une seule classe, souvent même dans une seule méthode. De telles méthodes tendent à être trop compliquées et à mêler la logique de sélection d'une stratégie avec son exécution.

Pour simplifier votre code, vous pouvez créer un groupe de classes, une pour chaque stratégie, puis définir une opération et la distribuer à travers ces classes. Chaque classe peut ainsi encapsuler une stratégie, réduisant considérablement le code. Vous devez aussi permettre au client qui utilise une stratégie d'en sélectionner une. Ce code de sélection peut être complexe même à l'issue de la refactorisation, mais vous devriez pouvoir le réduire jusqu'à ce qu'il ressemble presque à du pseudo-code décrivant la sélection d'une stratégie dans le domaine du problème.

Typiquement, un client conserve la stratégie sélectionnée dans une variable contextuelle. L'exécution de la stratégie revient ainsi simplement à transmettre au contexte l'appel de l'opération stratégique, en utilisant le polymorphisme pour exécuter la stratégie appropriée. En encapsulant les stratégies alternatives dans des classes séparées implémentant chacune une opération commune, le pattern STRATEGY permet de créer un code clair et simple qui modélise une famille d'approches pour résoudre un problème.

COMMAND

Le moyen classique de déclencher l'exécution d'une méthode est de l'appeler. Il arrive souvent néanmoins que vous ne puissiez pas contrôler le moment précis ou le contexte de son exécution. Dans de telles situations, vous pouvez l'encapsuler dans un objet. En stockant les informations nécessaires à l'invocation d'une méthode dans un objet, vous pouvez la passer en tant que paramètre, permettant ainsi à un client ou un service de déterminer quand l'invoquer.

L'objectif du pattern COMMAND est d'encapsuler une requête dans un objet.

Un exemple classique : commandes de menus

Les kits d'outils qui supportent des menus appliquent généralement le pattern COMMAND. Chaque élément de menu s'accompagne d'un objet qui sait comment se comporter lorsque l'utilisateur clique dessus. Cette conception permet de séparer la logique GUI de l'application. La bibliothèque Swing adopte cette approche, vous permettant d'associer un `ActionListener` à chaque `JMenuItem`.

Comment faire pour qu'une classe appelle une de vos méthodes lorsque l'utilisateur clique ? Il faut pour cela recourir au polymorphisme, c'est-à-dire rendre le nom de l'opération fixe et laisser l'implémentation varier. Pour `JMenuItem`, l'opération est `actionPerformed()`. Lorsque l'utilisateur fait un choix, l'élément `JMenuItem` appelle la méthode `actionPerformed()` de l'objet qui s'est enregistré en tant que listener.

Exercice 24.1

Le fonctionnement des menus Java facilite l'application du pattern COMMAND mais ne vous demande pas d'organiser votre code en commandes. En fait, il est fréquent de développer une application dans laquelle un seul objet écoute tous les événements d'une interface GUI. Quel pattern cela vous évoque-t-il ?

- *Les solutions des exercices de ce chapitre sont données dans l'Annexe B.*

Lorsque vous développez une application Swing, vous pouvez enregistrer un seul listener pour tous les événement GUI, plus particulièrement lorsque les composants GUI interagissent. Toutefois, pour les menus, il ne s'agit généralement pas de l'approche à suivre. Si vous deviez utiliser un seul objet pour écouter les menus, il devrait déterminer pour chaque événement l'objet GUI qui l'a généré. Au lieu de cela, lorsque vous avez plusieurs éléments de menu qui donnent lieu à des actions indépendantes, il peut être préférable d'appliquer COMMAND.

Lorsqu'un utilisateur sélectionne un élément de menu, il invoque la méthode `actionPerformed()`. Lorsque vous créez l'élément, vous pouvez lui associer un `ActionListener`, avec une méthode `actionPerformed()` spécifique au comportement de la commande. Plutôt que de définir une nouvelle classe pour implémenter ce petit comportement, il est courant d'employer une classe anonyme.

Considérez la classe `Visualization2` du package `com.oozinoz.visualization`. Elle fournit une barre de menus avec un menu File (Fichier) qui permet à l'utilisateur d'enregistrer et de restaurer les visualisations d'une unité de production Oozinoz simulée. Ce menu comporte des éléments Save As... (Enregistrer sous...) et Restore From... (Restaurer à partir de...). Le code qui crée ces éléments enregistre des listeners qui attendent la sélection de l'utilisateur. Ces listeners implémentent la méthode `actionPerformed()` en appelant les méthodes `save()` et `load()` de la classe `Visualization2` :

```
package com.oozinoz.visualization;
import java.awt.event.*;
import javax.swing.*;
import com.oozinoz.ui.*;
public class Visualization2 extends Visualization {
    public static void main(String[] args) {
        Visualization2 panel = new Visualization2(UI.NORMAL);
        JFrame frame =
            SwingFacade.launch(panel, "Operational Model");
```

```
frame.setJMenuBar(panel.menus());
frame.setVisible(true);
}

public Visualization2(UI ui) {
    super(ui);
}

public JMenuBar menus() {
    JMenuBar menuBar = new JMenuBar();

    JMenu menu = new JMenu("File");
    menuBar.add(menu);

    JMenuItem menuItem = new JMenuItem("Save As...");
    menuItem.addActionListener(new ActionListener() {
        // Exercice !
    });
    menu.add(menuItem);

    menuItem = new JMenuItem("Restore From...");
    menuItem.addActionListener(new ActionListener() {
        // Exercice !
    });
    menu.add(menuItem);

    return menuBar;
}

public void save() { /* omis */ }
public void restore() { /* omis */ }
}
```

Exercice 24.2

Complétez le code des sous-classes anonymes de `ActionListener`, en remplaçant la méthode `actionPerformed()`. Notez que cette méthode attend un argument `ActionEvent`.

Lorsque vous associez des commandes à un menu, vous les placez dans un contexte fourni par un autre développeur : le framework de menus Java. Dans d'autres utilisations de COMMAND, vous aurez le rôle de développer le contexte dans lequel les commandes s'exécuteront. Par exemple, vous pourriez vouloir fournir un service de minutage qui enregistre la durée d'exécution des méthodes.

Emploi de **COMMAND** pour fournir un service

Supposez que vous vouliez permettre aux développeurs de connaître la durée d'exécution d'une méthode. Vous disposez d'une interface **Command** dont voici l'essence :

```
public abstract void execute();
```

Vous disposez également de la classe **CommandTimer** suivante :

```
package com.oozinoz.utility;

import com.oozinoz.robotInterpreter.Command;

public class CommandTimer {
    public static long time(Command command) {
        long t1 = System.currentTimeMillis();
        command.execute();
        long t2 = System.currentTimeMillis();
        return t2 - t1;
    }
}
```

Vous pourriez tester la méthode **time()** au moyen d'un test JUnit ressemblant à ce qui suit. Notez que ce test n'est pas exact car il peut échouer si le timer est irrégulier :

```
package app.command;

import com.oozinoz.robotInterpreter.Command;
import com.oozinoz.utility.CommandTimer;

import junit.framework.TestCase;
public class TestCommandTimer extends TestCase {
    public void testSleep() {
        Command doze = new Command() {
            public void execute() {
                try {
                    Thread.sleep(
                        2000 + Math.round(10 * Math.random()));
                } catch (InterruptedException ignored) {
                }
            }
        };
        long actual = // Exercice !
    }
}
```

```
        long expected = 2000;
        long delta = 5;
        assertTrue(
            "Devrait être " + expected + " +/- " + delta + " ms",
            expected - delta <= actual
            && actual <= expected + delta);
    }
}
```

Exercice 24.3

Complétez les instructions d'assignation qui définissent la valeur de `actual` de manière que la commande `doze` soit minutée.

Hooks

Le Chapitre 21, consacré au pattern TEMPLATE METHOD, a introduit la presse à étoiles Aster, une machine intelligente qui inclut du code s'appuyant sur ce pattern. Le code de cette machine vous permet de remplacer une méthode qui marque un moule comme étant incomplet s'il est en cours de traitement au moment où elle est arrêtée.

La classe `AsterStarPress` est abstraite, vous demandant de dériver une sous-classe contenant une méthode `markMoldIncomplete()`. La méthode `shutDown()` de `AsterStarPress` utilise cette méthode pour garantir que l'objet du domaine sait que le moule est incomplet :

```
public void shutdown() {
    if (inProcess()) {
        stopProcessing();
        markMoldIncomplete(currentMoldID);
    }
    usherInputMolds();
    dischargePaste();
    flush();
}
```

Il peut vous sembler peu commode d'étendre `AsterStarPress` avec une classe que vous devez introduire dans l'ordinateur de bord de la presse. Supposez que vous demandiez aux développeurs de chez Aster de fournir le hook sous une forme différente, en utilisant le pattern COMMAND. La Figure 24.1 illustre une commande Hook

que la classe `AsterStarPress` peut utiliser, vous permettant de paramétriser la presse en cours d'exécution.

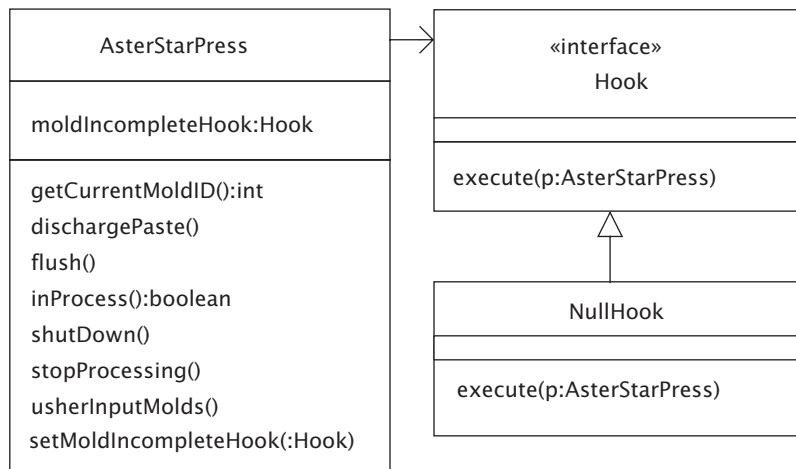


Figure 24.1

Une classe peut fournir un hook, c'est-à-dire un moyen d'insérer du code personnalisé, en invoquant une certaine commande à un point précis dans une procédure.

Dans la classe `AsterStarPress` originale, la méthode `shutDown()` s'appuyait sur une étape devant être définie par des sous-classes. Dans la nouvelle conception, cette méthode utilise un hook pour exécuter le code client après que le traitement a été interrompu mais avant la fin du processus d'arrêt :

```

public void shutDown() {
    if (inProcess()) {
        stopProcessing();
        // Exercice !
    }
    usherInputMolds();
    dischargePaste();
    flush();
}

```

Exercice 24.4

Complétez le code de la nouvelle méthode `shutDown()`.

Cet exemple est représentatif d'un autre pattern, **NULL OBJECT** [Woolf 1998], qui est seulement un peu moins connu que ceux décrits dans *Design Patterns*. Ce pattern permet d'éviter d'avoir à vérifier l'éventualité d'un pointeur `null` en introduisant un objet par défaut qui est sans effet (voir *Refactoring* [Fowler et al. 1999] pour une explication de la façon d'appliquer ce pattern à votre code). Le pattern **COMMAND** représente une conception alternative à **TEMPLATE METHOD** pour les hooks, et est semblable en termes d'objectif, ou de structure, à plusieurs autres patterns.

COMMAND en relation avec d'autres patterns

COMMAND ressemble au pattern **INTERPRETER**, ces deux patterns étant comparés dans le prochain chapitre. Il ressemble aussi à un pattern dans lequel un client sait quand une action est requise mais ne sait pas exactement quelle opération appeler.

Exercice 24.5

Quel pattern convient dans la situation où un client sait *quand* créer un objet mais ne sait pas quelle classe instancier ?

Outre des similitudes avec d'autres patterns, **COMMAND** collabore souvent également avec d'autres patterns. Par exemple, vous pourriez combiner **COMMAND** et **MEDIATOR** dans une conception MVC. Le Chapitre 19, consacré au pattern **MEMENTO**, en donne un exemple. La classe **Visualization** gère la logique de contrôle GUI mais confie à un médiateur la logique relative au modèle. Par exemple, cette classe utilise le code suivant pour effectuer une initialisation paresseuse de son bouton Undo :

```
protected JButton undoButton() {
    if (undoButton == null) {
        undoButton = ui.createButtonCancel();
        undoButton.setText("Undo");
        undoButton.setEnabled(false);
        undoButton.addActionListener(mediator.undoAction());
    }
    return undoButton;
}
```

Ce code applique **COMMAND**, introduisant une méthode `undo()` dans une instance de la classe `ActionListener`. Il applique également **MEDIATOR**, laissant un objet central servir de médiateur pour les événements appartenant à un modèle objet sous-jacent.

Pour que la méthode `undo()` fonctionne, le code médiateur doit restaurer une version antérieure de l'unité de production simulée, offrant l'opportunité d'appliquer un autre pattern qui accompagne souvent **COMMAND**.

Exercice 24.6

Quel pattern permet d'assurer le stockage et la restauration de l'état d'un objet ?

Résumé

Le pattern **COMMAND** sert à encapsuler une requête dans un objet, vous permettant de gérer des appels de méthodes en tant qu'objets, les passant et les invoquant lorsque le moment ou les conditions sont appropriés. Un exemple classique de l'intérêt de ce pattern a trait aux menus. Les éléments de menu savent *quand* exécuter une action mais ne savent pas quelle méthode appeler. **COMMAND** permet de paramétriser un menu avec des appels de méthodes correspondant aux options qu'il contient.

COMMAND peut aussi être utilisé pour permettre l'exécution de code client dans le contexte d'un service. Un service exécute souvent du code avant et après l'invocation du code client. Outre le fait de contrôler le moment ou le contexte d'exécution d'une méthode, ce pattern peut offrir un mécanisme commode pour fournir des hooks, permettant à un code client optionnel de s'exécuter dans le cadre d'un algorithme.

Un autre aspect fondamental de **COMMAND** est qu'il entretient plusieurs relations intéressantes avec d'autres patterns. Il peut constituer une alternative à **TEMPLATE METHOD** et peut aussi souvent collaborer avec **MEDIATOR** et **MEMENTO**.

INTERPRETER

A l'instar du pattern **C**OMMAND, le pattern **I**NTERPRETER produit un objet exécutable. Ces deux patterns diffèrent en ce que **I**NTERPRETER implique la création d'une hiérarchie de classes dans laquelle chaque classe implémente, ou *interprète*, une opération commune de sorte qu'elle corresponde au nom de la classe. A cet égard, **I**NTERPRETER est semblable aux patterns **S**TATE et **SRATEGY. Dans ces trois patterns, une opération commune apparaît dans une collection de classes, chacune d'elles implémentant l'opération de manière différente.**

Le pattern **I**NTERPRETER ressemble aussi au pattern **C**OMPOSITE, lequel définit une interface commune pour des éléments individuels ou des groupes d'éléments. **C**OMPOSITE ne requiert pas des moyens différents de former des groupes, bien qu'il le permette. Par exemple, la hiérarchie **P**rocess**C**omponent du Chapitre 5, consacré au pattern **C**OMPOSITE, autorise des séquences et des alternances de flux de processus. Dans **I**NTERPRETER, l'idée qu'il y ait différents types de compositions est essentielle (un **I**NTERPRETER est souvent placé au-dessus d'une structure **C**OMPOSITE). La façon dont une classe compose d'autres composants aide à définir comment une classe **I**NTERPRETER implémente une opération.

INTERPRETER n'est pas un pattern facile à comprendre. Vous pourriez avoir besoin de réviser le pattern **C**OMPOSITE car nous nous y référerons dans ce chapitre.

L'objectif du pattern **INTERPRETER est de vous permettre de composer des objets exécutables d'après un ensemble de règles de composition que vous définissez.**

Un exemple de INTERPRETER

Les robots qu’Oozinoz utilise pour déplacer des produits dans une ligne de traitement comportent un **interpréteur** qui contrôle le robot mais dispose d’un contrôle limité sur les machines de la ligne. Vous pourriez voir les interpréteurs comme se destinant aux langages de programmation, mais le cœur du pattern **INTERPRETER** est constitué d’une collection de classes qui permettent la composition d’instructions. L’interpréteur des robots d’Oozinoz consiste en une hiérarchie de classes qui encapsulent des commandes de robot. La hiérarchie comporte une classe abstraite **Command** en son sommet et inclut à tous les niveaux une opération **execute()**. La Figure 25.1 présente la classe **Robot** et deux des commandes supportées par l’interpréteur.

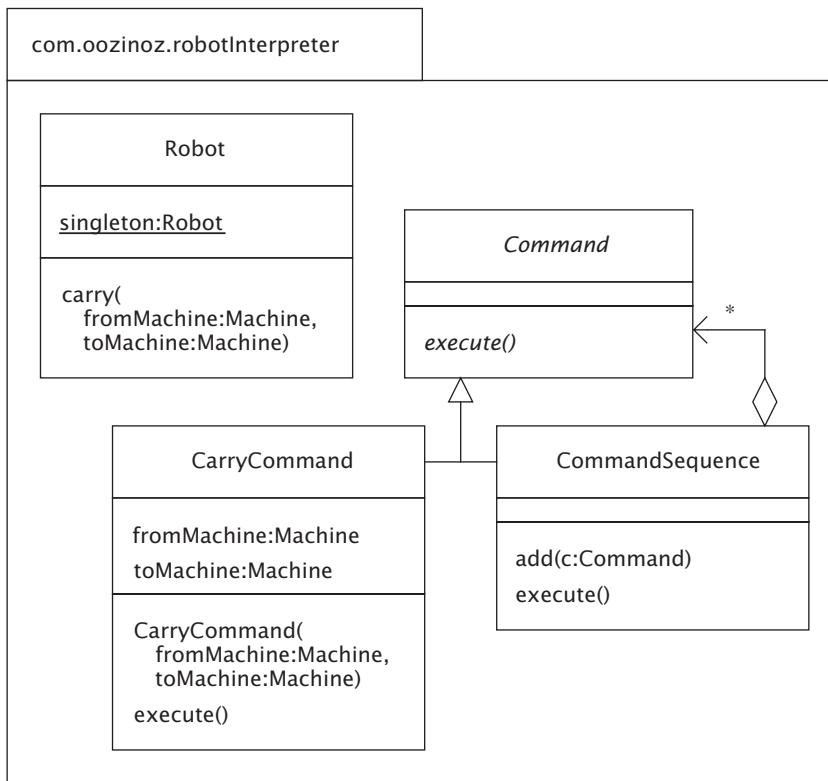


Figure 25.1

Une hiérarchie interpréteur supporte la programmation lors de l’exécution d’un robot d’usine.

Cette figure pourrait suggérer que le pattern COMMAND est présent dans cette conception, avec une classe Command au sommet de la hiérarchie. Toutefois, ce pattern a pour objectif d'encapsuler une méthode dans un objet, ce que ne fait pas ici la hiérarchie Command. Au lieu de cela, cette conception requiert que les sous-classes de Command réinterprètent l'opération `execute()`, ce qui est l'objectif du pattern INTERPRETER : vous permettre de composer des objets exécutables.

Une hiérarchie INTERPRETER typique inclurait plus de deux sous-classes et étendrait brièvement la hiérarchie Command. Les deux sous-classes de la Figure 25.1 suffisent néanmoins pour un exemple initial :

```
package app.interpreter;
import com.oozinoz.machine.*;
import com.oozinoz.robotInterpreter.*;

public class ShowInterpreter {
    public static void main(String[] args) {
        MachineComposite dublin = OozinozFactory.dublin();
        ShellAssembler assembler =
            (ShellAssembler) dublin.find("ShellAssembler:3302");
        StarPress press = (StarPress) dublin.find("StarPress:3404");
        Fuser fuser = (Fuser) dublin.find("Fuser:3102");

        assembler.load(new Bin(11011));
        press.load(new Bin(11015));

        CarryCommand carry1 = new CarryCommand(assembler, fuser);
        CarryCommand carry2 = new CarryCommand(press, fuser);

        CommandSequence seq = new CommandSequence();
        seq.add(carry1);
        seq.add(carry2);

        seq.execute();
    }
}
```

Ce code de démonstration fait qu'un robot d'usine déplace deux caisses de produits des machines opérationnelles vers un tampon de déchargement. Il fonctionne avec un composite de machines retourné par la méthode `dublin()` de la classe Oozinoz-Factory. Ce modèle de données représente une unité de production prévue pour un nouveau site à Dublin en Irlande. Le code localise trois machines au sein de l'usine, charge les caisses de produits sur deux d'entre elles, puis crée des commandes à

partir de la hiérarchie Command. La dernière instruction du programme appelle la méthode execute() d'un objet CommandSequence pour que le robot exécute les actions contenues dans la commande seq.

Un objet CommandSequence interprète l'opération execute() en transmettant l'appel à chaque sous-commande :

```
package com.oozinoz.robotInterpreter;

import java.util.ArrayList;
import java.util.List;

public class CommandSequence extends Command {
    protected List commands = new ArrayList();

    public void add(Command c) {
        commands.add(c);
    }

    public void execute() {
        for (int i = 0; i < commands.size(); i++) {
            Command c = (Command) commands.get(i);
            c.execute();
        }
    }
}
```

La classe CarryCommand interprète l'opération execute() en interagissant avec le robot pour déplacer une caisse d'une machine vers une autre :

```
package com.oozinoz.robotInterpreter;
import com.oozinoz.machine.Machine;

public class CarryCommand extends Command {
    protected Machine fromMachine;
    protected Machine toMachine;

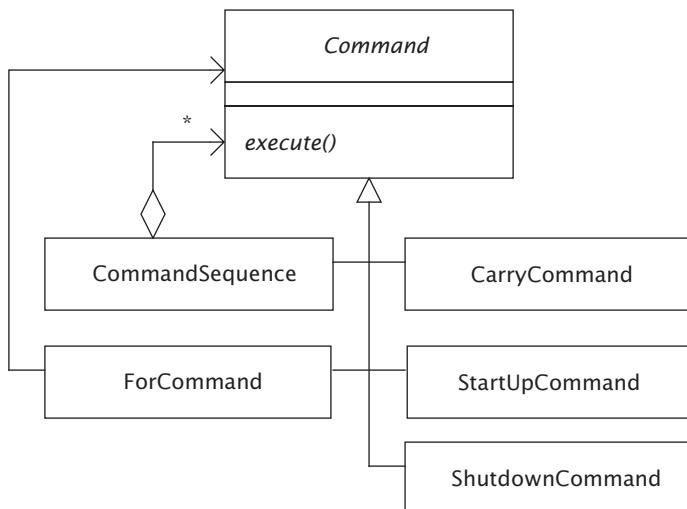
    public CarryCommand(
        Machine fromMachine, Machine toMachine) {
        this.fromMachine = fromMachine;
        this.toMachine = toMachine;
    }

    public void execute() {
        Robot.singleton.carry(fromMachine, toMachine);
    }
}
```

La classe `CarryCommand` a été conçue pour fonctionner spécifiquement dans le domaine d'une ligne de production contrôlée par des robots. On peut facilement imaginer d'autres classes spécifiques à un domaine, telles qu'une classe `StartUpCommand` ou `ShutdownCommand` pour contrôler les machines. Il serait également utile d'avoir une classe `ForCommand` qui exécute une commande à travers un ensemble de machines. La Figure 25.2 illustre ces extensions de la hiérarchie `Command`.

Figure 25.2

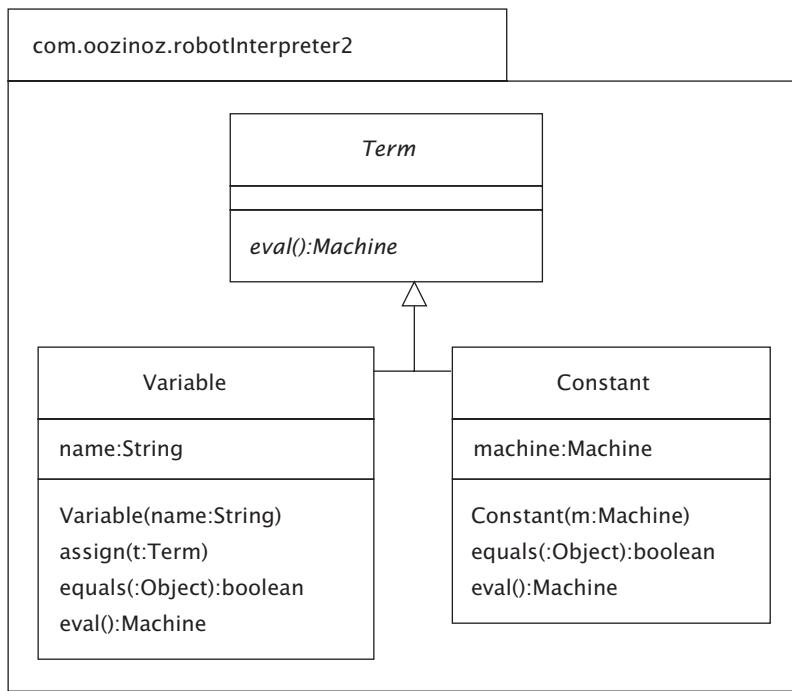
Le pattern INTERPRETER permet à plusieurs sous-classes de réinterpréter la signification d'une opération commune.



Une partie de la conception de la classe `ForCommand` apparaît claire d'emblée. Le constructeur de cette classe accepterait vraisemblablement une collection de machines et un objet `COMMAND` qui serait exécuté en tant que corps d'une boucle `for`. La partie la plus délicate est la liaison de la boucle et du corps. Java 5 possède une instruction `for` étendue qui établit une variable recevant une nouvelle valeur chaque fois que le corps est exécuté. Nous émulerons cette approche. Considérez l'instruction suivante :

```
for (Command c: commands)
    c.execute();
```

Java associe l'identifiant `c` déclaré par l'instruction `for` à la variable `c` du corps de la boucle. Pour créer une classe `INTERPRETER` qui émule cela, nous avons besoin d'un mécanisme pour gérer et évaluer des variables. La Figure 25.3 présente une hiérarchie `Term` qui sert à cela.

**Figure 25.3**

La hiérarchie `Term` fournit des variables pouvant représenter des machines.

La hiérarchie `Term` est semblable à la hiérarchie `Command` en ce qu'une certaine opération, en l'occurrence `eval()`, apparaît à tous les niveaux. Vous pourriez penser que cette hiérarchie est elle-même un exemple de `INTERPRETER`, malgré l'absence de classes de composition, telles que `CommandSequence`, qui accompagnent généralement ce pattern.

Cette hiérarchie permet de nommer des machines individuelles en tant que constantes et d'assigner des variables à ces constantes ou à d'autres variables. Elle apporte aussi plus de souplesse aux classes `INTERPRETER` spécifiques à un domaine. Par exemple, le code de `StartUpCommand` peut être conçu pour fonctionner avec un objet `Term` plutôt qu'avec une machine spécifique :

```

package com.oozinoz.robotInterpreter2;
import com.oozinoz.machine.Machine;

public class StartUpCommand extends Command {
    protected Term term;
}

```

```
public StartUpCommand(Term term) {
    this.term = term;
}

public void execute() {
    Machine m = term.eval();
    m.startup();
}
```

De même, pour ajouter plus de souplesse à la classe CarryCommand, nous pouvons la modifier pour qu'elle fonctionne avec des objets Term :

```
package com.oozinoz.robotInterpreter2;

public class CarryCommand extends Command {
    protected Term from;
    protected Term to;

    public CarryCommand(Term fromTerm, Term toTerm) {
        from = fromTerm;
        to = toTerm;
    }

    public void execute() {
        Robot.singleton.carry(from.eval(), to.eval());
    }
}
```

Après avoir conçu la hiérarchie Command pour qu'elle fonctionne avec des objets Term, nous pouvons écrire la classe ForCommand de façon qu'elle définisse la valeur d'une variable et exécute une commande body dans une boucle :

```
package com.oozinoz.robotInterpreter2;

import java.util.List;
import com.oozinoz.machine.Machine;
import com.oozinoz.machine.MachineComponent;
import com.oozinoz.machine.MachineComposite;

public class ForCommand extends Command {
    protected MachineComponent root;
    protected Variable variable;
    protected Command body;

    public ForCommand(
        MachineComponent mc, Variable v, Command body) {
        this.root = mc;
        this.variable = v;
        this.body = body;
    }
}
```

```
public void execute() {
    execute(root);
}

private void execute(MachineComponent mc) {
    if (mc instanceof Machine) {
        // Exercice !
        return;
    }

    MachineComposite comp = (MachineComposite) mc;
    List children = comp.getComponents();
    for (int i = 0; i < children.size(); i++) {
        MachineComponent child =
            (MachineComponent) children.get(i);
        execute(child);
    }
}
```

Le code `execute()` de la classe `ForCommand` recourt au transtypage (*casting*) pour parcourir un arbre de composant-machine. Le Chapitre 28, sur le pattern `ITERATOR`, présente des techniques plus rapides et plus élégantes pour explorer un composite. Pour le pattern `INTERPRETER`, l'important est d'interpréter correctement la requête `execute()` pour chaque nœud de l'arbre.

Exercice 25.1

Complétez le code de la méthode `execute()` de la classe `ForCommand`.

▪ Les solutions des exercices de ce chapitre sont données dans l'Annexe B.

La classe `ForCommand` nous permet de commencer à composer des programmes, ou scripts, de commandes pour l'unité de production. Voici par exemple un programme qui compose un objet interpréteur qui arrête toutes les machines dans l'unité :

```
package app.interpreter;

import com.oozinoz.machine.*;
import com.oozinoz.robotInterpreter2.*;

class ShowDown {
    public static void main(String[] args) {
```

```

MachineComposite dublin = OozinozFactory.dublin();
Variable v = new Variable("machine");
Command c = new ForCommand(
    dublin, v, new ShutDownCommand(v));
c.execute();
}
}

```

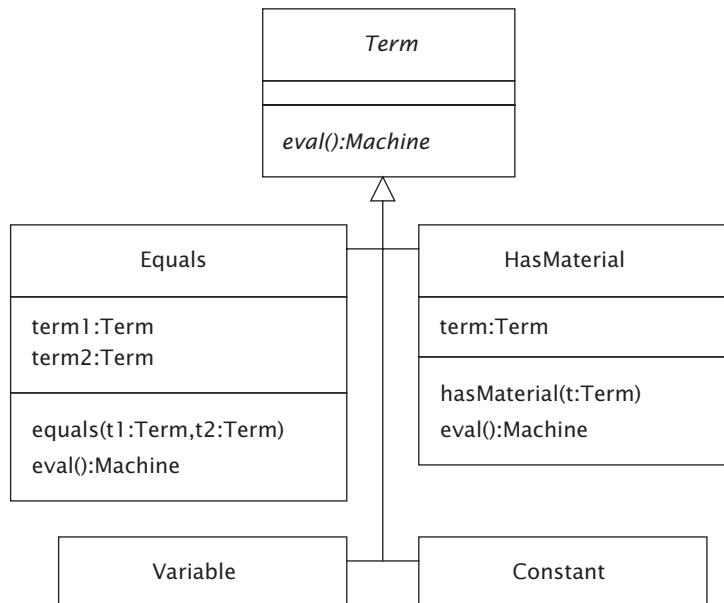
Lorsque ce programme appelle la méthode `execute()`, l'objet `ForCommand` l'interprète en traversant le composant-machine fourni et, pour chaque machine :

- Il définit la valeur de la variable `v`.
- Il invoque l'opération `execute()` de l'objet `ShutDownCommand` fourni.

Si nous ajoutons des classes qui contrôlent le flux logique, telles qu'une classe `IfCommand` et une classe `WhileCommand`, nous pouvons créer un interpréteur riche en fonctionnalités. Ces classes doivent disposer d'un moyen pour modéliser une condition booléenne. Nous pourrions par exemple avoir besoin de modéliser si une variable machine est égale à une certaine machine. A cet effet, nous pourrions introduire une nouvelle hiérarchie d'objets `Term`, bien qu'il soit plus simple d'emprunter une idée du langage C : nous disons que `null` signifie faux et que tout le reste signifie vrai. Nous pouvons ainsi étendre la hiérarchie `Term`, comme le montre la Figure 25.4.

Figure 25.4

La hiérarchie `Term` inclut des classes qui modélisent des conditions booléennes.



La classe Equals compare deux termes et retourne null pour signifier que la condition d'égalité est fausse. Une conception raisonnable serait de faire en sorte que la méthode eval() de cette classe retourne un de ses termes si l'égalité est vraie, comme suit :

```
package com.oozinoz.robotInterpreter2;
import com.oozinoz.machine.Machine;
public class Equals extends Term {
    protected Term term1;
    protected Term term2;

    public Equals(Term term1, Term term2) {
        this.term1 = term1;
        this.term2 = term2;
    }

    public Machine eval() {
        Machine m1 = term1.eval();
        Machine m2 = term2.eval();
        return m1.equals(m2) ? m1 : null;
    }
}
```

La classe HasMaterial étend l'idée d'une valeur de classe booléenne à un exemple spécifique à un domaine :

```
package com.oozinoz.robotInterpreter2;

import com.oozinoz.machine.Machine;

public class HasMaterial extends Term {
    protected Term term;

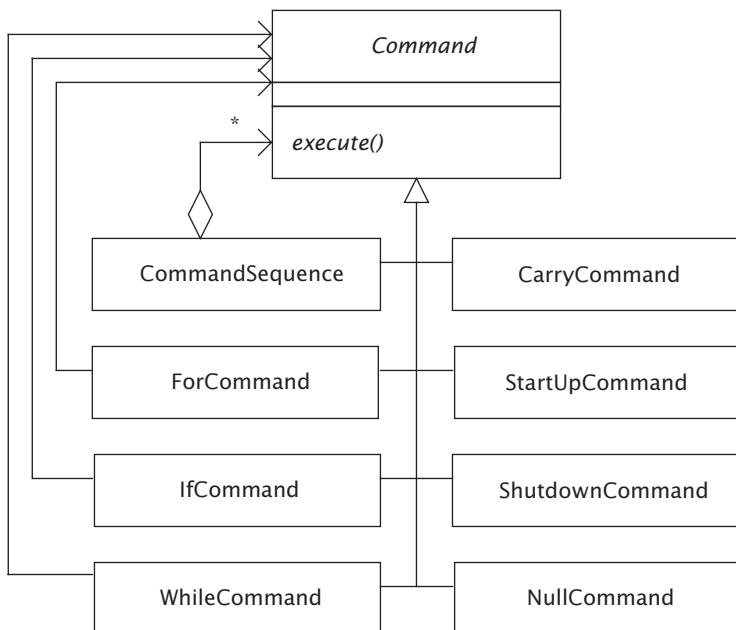
    public HasMaterial(Term term) {
        this.term = term;
    }

    public Machine eval() {
        Machine m = term.eval();
        return m.hasMaterial() ? m : null;
    }
}
```

Maintenant que nous avons ajouté l'idée de termes booléens à notre package interpréteur, nous pouvons ajouter des classes de contrôle de flux, comme illustré Figure 25.5.

Figure 25.5

Nous pouvons obtenir un interpréteur plus riche en fonctionnalités en ajoutant à sa hiérarchie des classes de contrôle de flux.



La classe **NullCommand** est utile pour les cas où nous avons besoin d'une commande qui ne fait rien, comme lorsque la branche **else** d'une instruction **if** est vide :

```

package com.oozinoz.robotInterpreter2;
public class NullCommand extends Command {
    public void execute() {
    }
}

package com.oozinoz.robotInterpreter2;

public class IfCommand extends Command {
    protected Term term;
    protected Command body;
    protected Command elseBody;

    public IfCommand(
        Term term, Command body, Command elseBody) {
        this.term = term;
        this.body = body;
        this.elseBody = elseBody;
    }
}
  
```

```
public void execute() {  
    // Exercice !  
}  
}
```

Exercice 25.2

Complétez le code de la méthode `execute()` de la classe `IfCommand`.

Exercice 25.3

Ecrivez le code de la classe `WhileCommand`.

Nous pourrions utiliser la classe `WhileCommand` avec un interpréteur qui décharge une presse à étoiles :

```
package app.interpreter;  
  
import com.oozinoz.machine.*;  
import com.oozinoz.robotInterpreter2.*;  
  
public class ShowWhile {  
    public static void main(String[] args) {  
        MachineComposite dublin = OozinozFactory.dublin();  
        Term starPress = new Constant(  
            (Machine) dublin.find("StarPress:1401"));  
        Term fuser = new Constant(  
            (Machine) dublin.find("Fuser:1101"));  
  
        starPress.eval().load(new Bin(77));  
        starPress.eval().load(new Bin(88));  
  
        WhileCommand command = new WhileCommand(  
            new HasMaterial(starPress),  
            new CarryCommand(starPress, fuser));  
        command.execute();  
    }  
}
```

L'objet `command` est un interpréteur qui interprète `execute()` pour signifier de décharger toutes les caisses de la presse 1401.

Exercice 25.4

Fermez ce livre et expliquez brièvement la différence entre COMMAND et INTERPRETER.

Nous pourrions ajouter d'autres classes à la hiérarchie de l'interpréteur pour avoir plus de types de contrôle ou pour des tâches relevant d'autres domaines. Nous pourrions aussi étendre la hiérarchie `Term`. Par exemple, il pourrait être utile de disposer d'une sous-classe `Term` qui localise un tampon de déchargement proche d'une autre machine.

Les utilisateurs des hiérarchies `Command` et `Term` peuvent composer des "programmes" d'exécution complexes. Par exemple, il ne serait pas trop difficile de créer un objet qui, lorsqu'il s'exécute, décharge tous les produits de toutes les machines, à l'exception des tampons de déchargement. Voici le pseudo-code d'un tel programme :

```
for (m dans usine)
    if (not (m est tamponDechargement))
        td = chercheMethDecharg pour m
        while (m contientProduit)
            transporte (m, td)
```

Si nous écrivions du code Java pour accomplir ces tâches, le résultat serait plus volumineux et moins simple que le pseudo-code. Aussi, pourquoi ne pas transformer ce dernier en du code réel en créant un analyseur syntaxique capable de lire un langage spécifique à un domaine pour manipuler les produits de l'usine et de créer des objets interpréteur à notre place ?

Interpréteurs, langages et analyseurs syntaxiques

Le pattern `INTERPRETER` spécifie comment les interpréteurs fonctionnent mais pas comment il faut les instancier ou les composer. Dans ce chapitre, vous avez créé des interpréteurs manuellement, en écrivant directement les lignes de code Java. Une approche plus courante est d'utiliser un **analyseur syntaxique** (*parser*), c'est-à-dire un objet qui peut reconnaître du texte et décomposer sa structure à partir d'un ensemble de règles pour le mettre dans une forme adaptée à un traitement subséquent. Vous pourriez par exemple écrire un analyseur qui crée un interpréteur de commandes machine correspondant au pseudo-code présenté plus haut.

Au moment de la rédaction du présent livre, il existait peu d'outils pour analyser le code Java et seulement quelques ouvrages sur le sujet. Pour déterminer si un support plus large a été développé depuis, recherchez sur le Web la chaîne "Java parser tools". La plupart des kits d'outils d'analyse incluent un générateur de parser. Pour l'utiliser, vous devez employer une syntaxe spéciale décrivant la *grammaire* de votre langage, et l'outil générera un analyseur à partir de votre description. Cet analyseur reconnaîtra ensuite les instances de votre langage. Vous pouvez sinon écrire vous-même un analyseur à caractère général en appliquant le pattern INTERPRETER. L'ouvrage *Building Parsers with JavaTM* [Metsker 2001] explique cette technique, avec des exemples en Java.

Résumé

Le pattern INTERPRETER permet de composer des objets exécutables à partir d'une hiérarchie de classes que vous créez. Chaque classe implémente une opération commune qui possède habituellement un nom générique, tel que `execute()`. Bien que les exemples de ce chapitre ne le montrent pas, cette méthode reçoit souvent un objet "contexte" additionnel qui stocke un état important.

Le nom de chaque classe reflète généralement la façon dont elle implémente, ou interprète, l'opération commune. Chacune d'elles définit un moyen de composer des commandes ou bien représente une commande terminale qui entraîne une action.

Les interpréteurs s'accompagnent souvent d'une conception supportant l'introduction de variables et d'expressions booléennes ou arithmétiques. Ils collaborent souvent aussi avec un analyseur syntaxique pour créer un petit langage qui simplifie la génération de nouveaux objets interpréteur.

V

Patterns d'extension

Introduction aux extensions

Lorsque vous programmez en Java, vous ne partez pas de zéro mais "héritez" de toute la puissance des bibliothèques de classes de ce langage. Vous héritez aussi habituellement du code de vos prédecesseurs et de vos collègues. Lorsque vous ne réorganisez ou n'améliorez pas le code existant, vous l'étendez. Vous pourriez dire que la programmation en Java *est* une extension. Il vous est peut-être déjà arrivé d'hériter d'une base de code dont la qualité était médiocre. Mais le code que vous ajoutez est-il réellement meilleur ? La réponse est parfois subjective. Ce chapitre passe en revue certains principes du développement orienté objet qui vous permettront d'évaluer la qualité de votre travail.

Outre les techniques classiques d'extension d'une base de code, vous pourriez appliquer des patterns de conception pour ajouter de nouveaux comportements. Après avoir exposé les principes de la conception orientée objet dans le cadre d'un développement ordinaire, cette section revoit certains patterns contenant un élément d'extension et introduit ceux non encore abordés.

Principes de la conception orientée objet

Les ponts de pierre existent depuis plusieurs milliers d'années, ce qui a laissé à l'homme le temps de développer des principes de conception bien éprouvés. La programmation orientée objet existe, elle, depuis peut-être cinquante ans, aussi n'est-il pas surprenant que ses principes de conception en soient à un stade moins avancé. Nous disposons toutefois d'excellents forums sur les principes reconnus actuellement, l'un des meilleurs étant Portland Pattern Repository à l'adresse www.c2.com [Cunningham]. Vous trouverez sur ce site les principes les plus efficaces pour évaluer des conceptions OO. L'un d'eux, notamment, est le principe de substitution de Liskov.

Le principe de substitution de Liskov

Les nouvelles classes devraient être des extensions logiques et cohérentes de leurs super-classes. Un compilateur Java garantit un certain niveau de cohérence, mais nombre des principes de cohérence lui échappent. Une règle qui peut vous aider à améliorer vos conceptions est le **principe de substitution de Liskov**, ou LSP (*Liskov Substitution Principle*) [Liskov 1987], qui peut être paraphrasé comme suit :

Une instance d'une classe devrait fonctionner comme une instance de sa super-classe.

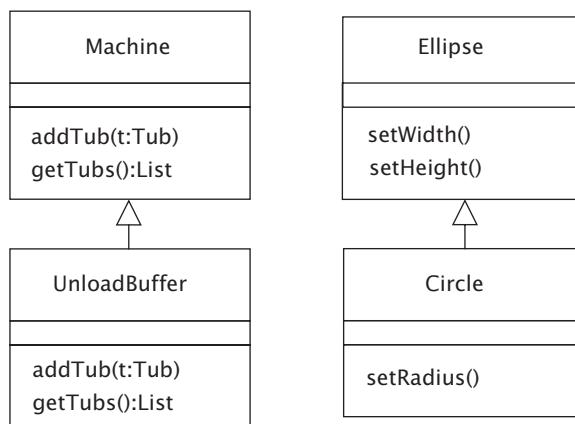
Une conformité basique avec ce principe est intégrée aux langages OO, tels que Java. Par exemple, il est valide de se référer à un objet `UnloadBuffer` (tampon de déchargement) en tant que `Machine` puisque `UnloadBuffer` est une sous-classe de `Machine` :

```
Machine m = new UnloadBuffer(3501);
```

Certains aspects de la conformité avec LSP font appel à l'intelligence humaine, ou en tout cas à plus d'intelligence que n'en possèdent les compilateurs actuels. Considérez les hiérarchies de classes de la Figure 26.1.

Figure 26.1

Ce diagramme s'applique aux questions suivantes : un tampon de déchargement est-il une machine ? Un cercle est-il une ellipse ?



Un tampon de déchargement est certainement une machine, mais modéliser ce fait dans une hiérarchie de classes peut donner lieu à des problèmes. Chez Oozinoz, toutes les machines, à l'exception des tampons de déchargement, peuvent recevoir un bac (*tub*) de produits chimiques et signaler les bacs qui se trouvent à côté d'elles. Aussi est-il utile de remonter ce comportement au niveau de la classe `Machine`.

Mais c'est une erreur que d'invoquer `addTub()` ou `getTubs()` sur un objet `UnloadBuffer`. Devrions-nous générer des exceptions si de tels appels avaient lieu ?

Supposez qu'un autre développeur écrive une méthode qui interroge toutes les machines d'une travée pour créer une liste complète de tous les bacs de produits présents dans cette travée. Lorsque ce code arrive à un tampon de déchargement, il rencontre une exception si la méthode `getTubs()` de la classe `UnloadBuffer` en génère une. Il s'agit d'une violation totale du principe de Liskov : si vous utilisez un objet `UnloadBuffer` en tant qu'objet `Machine`, votre programme peut planter. Au lieu de générer une exception, imaginez que nous décidions d'ignorer simplement les appels de `getTubs()` et `addTub()` dans la classe `UnloadBuffer`, ce qui représente toujours une violation de ce principe : si vous ajoutez un bac à une machine, le bac risque de disparaître.

Ces violations ne constituent pas toujours des erreurs de conception. Dans le cas des machines d'Oozinoz, il convient d'évaluer l'intérêt de placer dans la classe `Machine` des comportements qui s'appliquent à presque toutes les machines par rapport aux inconvénients liés au non-respect de LSP. L'important est de connaître ce principe et d'être capable de déterminer en quoi certaines considérations de conception justifient sa violation.

Exercice 26.1

Un cercle est certainement un cas spécial d'ellipse. Déterminez néanmoins si la relation des classes `Ellipse` et `Circle` dans la Figure 26.1 est une violation de LSP.

- *Les solutions des exercices de ce chapitre sont données dans l'Annexe B.*

La loi de Demeter

Vers la fin des années 80, des membres du Demeter Project à la Northeastern University de Boston ont tenté de codifier les règles garantissant la "bonne santé" d'un programme, leur attribuant le nom de **loi de Demeter**, ou LOD (*Law Of Demeter*). Karl Lieberherr et Ian Holland [1989] en ont donné une description détaillée dans un article intitulé "Assuring good style for object-oriented programs", qui dit ceci :

De manière informelle, la loi stipule que chaque méthode peut envoyer des messages uniquement à un ensemble limité d'objets : aux objets argument, à la pseudo-variable `[this]` et aux sous-parties immédiates de `[this]`.

L'article donne également une définition plus formelle, mais il est plus facile de donner des exemples de violations de cette loi que de tenter d'expliquer son propos.

Supposez que vous disposiez d'un objet `MaterialManager` avec une méthode qui reçoit un objet `Tub` en tant que paramètre. Les objets `Tub` possèdent une propriété `Location` qui retourne l'objet `Machine` représentant l'emplacement du bac. Dans la méthode de `MaterialManager`, vous avez besoin de savoir si la machine est active et disponible. Vous pourriez écrire le code suivant à cet effet :

```
if (tub.getLocation().isUp()) {  
    //...  
}
```

Ce code enfreint la loi de Demeter car il envoie un message à `tub.getLocation()`. L'objet `tub.getLocation()` n'est pas un paramètre, n'est pas `this` — l'objet `MaterialManager` dont la méthode est exécutée — et n'est pas non plus un attribut de `this`.

Exercice 26.2

Expliquez pourquoi l'expression `tub.getLocation().isUp()` peut être considérée comme étant de préférence à éviter.

Cet exercice pourrait banaliser la valeur de cette loi s'il suggérait qu'elle signifie seulement que les expressions de la forme `a.b.c` sont à éviter. En fait, Lieberherr et Holland lui donnent un sens plus large et répondent affirmativement à la question : "Existe-t-il une formule ou une règle permettant d'écrire des programmes orientés objet de bonne qualité ?" Je vous conseille de lire l'article original expliquant cette loi. A l'instar du principe de Liskov, elle vous aidera à développer de meilleurs programmes si vous connaissez les règles, savez les suivre, et savez aussi quand votre conception peut les enfreindre.

Vous pouvez constater qu'en suivant un ensemble de recommandations, vos extensions produisent automatiquement du code performant. Mais, aux yeux de nombreux programmeurs, le développement OO reste un art. L'extension efficace d'une base de code semble être le résultat d'un ensemble de pratiques élaborées par des artisans qui en sont encore à formuler et à codifier leur art. La **refactorisation** désigne l'emploi d'une collection d'outils de modification de code conçus pour améliorer la qualité d'une base de code sans changer sa fonction.

Elimination des erreurs potentielles

Vous pourriez penser que le principe de substitution de Liskov et la loi de Demeter vous empêcheront d'écrire du code médiocre. En fait, vous utiliserez plutôt ces règles pour identifier le code de mauvaise qualité et l'améliorer. C'est une pratique normale : écrire du code qui tourne puis le perfectionner en identifiant et en corrigeant les problèmes potentiels. Mais comment identifie-t-on ces problèmes ? Au moyen d'**indicateurs** (*code smell*). L'ouvrage *Refactoring: Improving the Design of Existing Code* [Fowler et al. 1999], en dénombre vingt-deux et décrit les refactorisations correspondantes.

Le présent livre a eu recours de nombreuses fois à la refactorisation pour réorganiser et améliorer du code existant en appliquant un pattern. Mais vous n'avez pas nécessairement besoin d'appliquer un pattern de conception lors de la refactorisation. Chaque fois que le code d'une méthode est susceptible de poser problème, il devrait être revu.

Exercice 26.3

Donnez un exemple d'une méthode pouvant être refactorisée sans enfreindre pour autant le principe de Liskov ou la loi de Demeter.

Au-delà des extensions ordinaires

L'objectif de beaucoup de patterns de conception, dont nombre d'entre eux ont déjà été couverts, a trait de près ou de loin à l'extension de comportements. Ces patterns clarifient souvent le rôle de deux développeurs. Par exemple, dans le pattern ADAPTER, un développeur peut fournir un service utile ainsi qu'une interface pour les objets qui veulent utiliser ce service.

En plus des patterns déjà couverts, trois autres patterns ont comme principal objectif d'étendre du code existant.

Si vous envisagez de

- Permettre aux développeurs de composer dynamiquement le comportement d'un objet

Appliquez le pattern

DECORATOR

<i>Si vous envisagez de</i>	<i>Appliquez le pattern</i>
• Offrir un moyen d'accéder aux éléments d'une collection de façon séquentielle	ITERATOR
• Permettre aux développeurs de définir une nouvelle opération pour une hiérarchie sans changer les classes qui la composent	VISITOR

Exercice 26.4

Complétez le tableau suivant, qui donne des exemples d'utilisation des patterns déjà abordés pour étendre le comportement d'une classe ou d'un objet.

<i>Exemple</i>	<i>Pattern à l'œuvre</i>
Le concepteur d'une simulation pyrotechnique établit une interface qui définit le comportement que doit présenter votre objet pour pouvoir prendre part à la simulation.	ADAPTER
Un kit d'outils permet de composer des objets exécutables lors de l'exécution.	?
?	TEMPLATE METHOD
?	COMMAND
Un générateur de code insère un comportement qui donne l'illusion qu'un objet s'exécutant sur une autre machine est local.	?
?	OBSERVER
Une conception permet de définir des opérations abstraites qui dépendent d'une interface spécifique et d'ajouter de nouveaux drivers qui satisfont aux exigences de cette interface.	?

Résumé

Ecrire du code revient souvent à étendre celui existant pour apporter de nouvelles fonctionnalités, puis à le réorganiser afin d'améliorer sa qualité. Il n'existe pas de technique infaillible pour évaluer la qualité d'un programme, mais certains principes de bonne conception OO ont néanmoins été définis.

Le principe de substitution de Liskov suggère qu'une instance d'une classe devrait fonctionner comme une instance de sa super-classe. Vous devriez connaître et être capable de justifier les violations de ce principe dans votre code. La loi de Demeter est un ensemble de règles qui aident à réduire les dépendances entre les classes et à clarifier le code.

Martin Fowler et al. [1999] ont élaboré une série d'indicateurs permettant d'identifier le code de qualité médiocre. Chaque indicateur donne lieu à une ou plusieurs refactorisations, certaines d'entre elles visant l'application d'un pattern de conception. Nombre de patterns servent de techniques pour clarifier, simplifier ou faciliter les extensions.

DECORATOR

Pour étendre une base de code, vous lui ajoutez normalement de nouvelles classes ou méthodes. Parfois, vous avez besoin de composer un objet avec un nouveau comportement lors de l'exécution. Le pattern INTERPRETER, par exemple, permet de générer un objet exécutable dont le comportement change radicalement selon la façon dont vous le composez. Dans certains cas, vous aurez besoin de pouvoir combiner plusieurs variations comportementales moins importantes et utiliserez pour cela le pattern DECORATOR.

L'objectif du pattern DECORATOR est de vous permettre de composer de nouvelles variations d'une opération lors de l'exécution.

Un exemple classique : flux d'E/S et objets Writer

La conception d'ensemble des flux d'entrée et de sortie dans les bibliothèques de classes Java constitue un exemple classique du pattern DECORATOR. Un **flux** (*stream*) est une série d'octets ou de caractères, tels ceux contenus dans un document. Dans Java, les classes d'écriture, ou **Writer**, représentent une façon de supporter les flux. Certaines de ces classes possèdent un constructeur qui accepte un objet **Writer**, ce qui signifie que vous pouvez créer un **Writer** à partir d'un **Writer**. Ce type de composition simple est la structure typique du pattern DECORATOR, qui est présent dans les classes d'écriture Java. Mais, comme nous le verrons, il ne faut pas beaucoup de code à DECORATOR pour nous permettre d'étendre grandement notre capacité à combiner des variations d'opérations de lecture et d'écriture.

Pour un exemple de DECORATOR dans Java, considérez le code suivant qui crée un petit fichier texte :

```
package app.decorator;
import java.io.*;

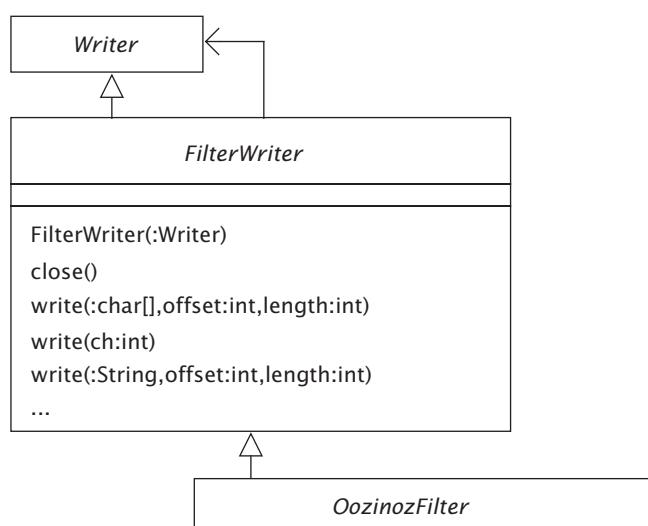
public class ShowDecorator {
    public static void main(String[] args)
        throws IOException {
        FileWriter file = new FileWriter("sample.txt");
        BufferedWriter writer = new BufferedWriter(file);
        writer.write("un petit exemple de texte");
        writer.newLine();
        writer.close();
    }
}
```

L'exécution de ce programme produit un fichier sample.txt qui contient une petite quantité de texte. Le code utilise un objet `FileWriter` pour créer un fichier, en enveloppant cet objet dans un objet `BufferedWriter`. Ce qu'il importe de retenir ici, c'est que nous composons un flux, `BufferedWriter`, à partir d'un autre flux, `FileWriter`.

Chez Oozinoz, le personnel de vente a besoin de mettre en forme des messages personnalisés à partir du texte stocké dans la base de données de produits. Ces messages n'utilisent pas des polices ou des styles très variés. Nous créerons pour cela un framework de décorateurs. Ces classes nous permettront de composer une grande variété de filtres de sortie.

Figure 27.1

La classe `OozinozFilter` est parente des classes qui mettent en forme les flux de caractères en sortie.



Pour développer une collection de classes de filtrage, il est utile de créer une classe abstraite qui définit les opérations que ces filtres doivent supporter. En sélectionnant des opérations qui existent déjà dans la classe `Writer`, vous pouvez créer presque sans effort une autre classe qui hérite tous ses comportements de cette classe. La Figure 27.1 illustre cette conception.

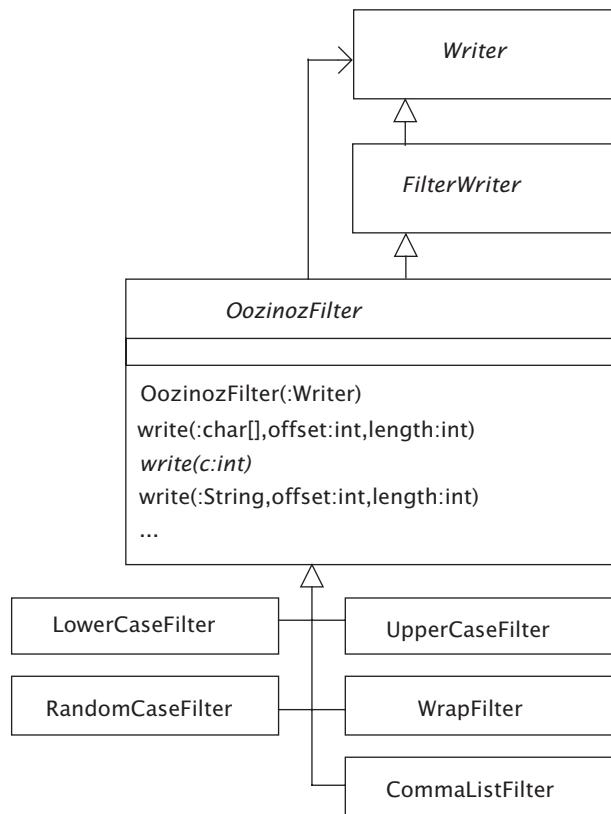
Notre super-classe de filtrage doit posséder plusieurs attributs essentiels pour pouvoir supporter des flux de sortie composable :

- Elle doit accepter dans son constructeur un objet `Writer`.
- Elle doit agir en tant que super-classe d'une hiérarchie de filtres.
- Elle doit fournir des implémentations par défaut de toutes les méthodes d'écriture sauf `write(:int)`.

La Figure 27.2 illustre cette conception.

Figure 27.2

Le constructeur de la classe `OozinozFilter` accepte une instance de n'importe quelle sous-classe de `Writer`.



La classe OozinozFilter répond aux exigences de conception en peu de lignes :

```
package com.oozinoz.filter;
import java.io.*;
public abstract class OozinozFilter extends FilterWriter {
    protected OozinozFilter(Writer out) {
        super(out);
    }

    public void write(char cbuf[], int offset, int length)
        throws IOException {
        for (int i = 0; i < length; i++)
            write(cbuf[offset + i]);
    }

    public abstract void write(int c) throws IOException;

    public void write(String s, int offset, int length)
        throws IOException {
        write(s.toCharArray(), offset, length);
    }
}
```

Ce code est tout ce dont nous avons besoin pour faire intervenir DECORATOR. Les sous-classes de OozinozFilter peuvent fournir de nouvelles implémentations de `write(:int)` qui modifient un caractère avant de le passer à la méthode `write(:int)` du flux sous-jacent. Les autres méthodes de OozinozFilter fournissent le comportement typiquement requis par les sous-classes. Cette classe laisse simplement les appels de `close()` et `flush()` à sa classe parent, `FilterWriter`. Elle interprète également `write(:char[])` par rapport à la méthode `write(:int)` qu'elle laisse abstraite.

A présent, il est aisé de créer et d'utiliser de nouveaux filtres de flux. Par exemple, le code suivant convertit le texte en minuscules :

```
package com.oozinoz.filter;
import java.io.*;

public class LowerCaseFilter extends OozinozFilter {
    public LowerCaseFilter(Writer out) {
        super(out);
    }

    public void write(int c) throws IOException {
        out.write(Character.toLowerCase((char) c));
    }
}
```

Voici un exemple de programme qui utilise un filtre de conversion en minuscules :

```
package app.decorator;

import java.io.IOException;
import java.io.Writer;

import com.oozinoz.filter.ConsoleWriter;
import com.oozinoz.filterLowerCaseFilter;

public class ShowLowerCase {
    public static void main(String[] args)
        throws IOException {
        Writer out = new ConsoleWriter();
        out = new LowerCaseFilter(out);
        out.write("Ce Texte doit être écrit TOUT en MiNusculeS !");
        out.close();
    }
}
```

Ce programme affiche "ce texte doit être écrit tout en minuscules !" sur la console.

Le code de la classe `UpperCaseFilter` est identique à celui de `LowerCaseFilter`, à l'exception de la méthode `write()`, que voici :

```
public void write(int c) throws IOException {
    out.write(Character.toUpperCase((char) c));
}
```

Le code de la classe `TitleCaseFilter` est un peu plus complexe puisqu'il doit garder trace des espaces :

```
package com.oozinoz.filter;
import java.io.*;

public class TitleCaseFilter extends OozinozFilter {
    boolean inWhite = true;

    public TitleCaseFilter(Writer out) {
        super(out);
    }

    public void write(int c) throws IOException {
        out.write(
            inWhite
                ? Character.toUpperCase((char) c)
                : Character.toLowerCase((char) c));
        inWhite = Character.isWhitespace((char) c)
            || c == ' ';
    }
}
```

La classe `CommaListFilter` insère une virgule entre des éléments :

```
package com.oozinoz.filter;

import java.io.IOException;
import java.io.Writer;

public class CommaListFilter extends OozinozFilter {
    protected boolean needComma = false;

    public CommaListFilter(Writer writer) {
        super(writer);
    }

    public void write(int c) throws IOException {
        if (needComma) {
            out.write(',');
            out.write(' ');
        }
        out.write(c);
        needComma = true;
    }

    public void write(String s) throws IOException {
        if (needComma)
            out.write(", ");
        out.write(s);
        needComma = true;
    }
}
```

Le rôle de ces filtres est le même : la tâche de développement consiste à remplacer les méthodes `write()` appropriées. Ces méthodes mettent en forme le flux de texte reçu puis le passent à un flux subordonné.

Exercice 27.1

Ecrivez le code de `RandomCaseFilter.java`.

▪ *Les solutions des exercices de ce chapitre sont données dans l'Annexe B.*

Le code de la classe `WrapFilter` est beaucoup plus complexe que les autres filtres. Il aligne son résultat au centre et doit donc mettre en tampon et compter les caractères avant de les passer à son flux subordonné. Vous pouvez examiner ce code en le

téléchargeant à partir de **www.oozinoz.com** (voir l'Annexe C pour les instructions de téléchargement du code).

Le constructeur de `WrapFilter` accepte un objet `Writer` ainsi qu'un paramètre de largeur lui indiquant quand aller à la ligne. Vous pouvez combiner ce filtre et d'autres filtres pour créer une variété d'effets. Par exemple, le programme suivant aligne au centre le texte d'un fichier en entrée, en insérant des retours à la ligne et en mettant en capitale la première lettre de chaque mot :

```
package app.decorator;
import java.io.*;
import com.oozinoz.filter.TitleCaseFilter;
import com.oozinoz.filter.WrapFilter;

public class ShowFilters {
    public static void main(String args[])
        throws IOException {
        BufferedReader in = new BufferedReader(
            new FileReader(args[0]));
        Writer out = new FileWriter(args[1]);
        out = new WrapFilter(new BufferedWriter(out), 40);
        out = new TitleCaseFilter(out);

        String line;
        while ((line = in.readLine()) != null)
            out.write(line + "\n");
        out.close();
        in.close();
    }
}
```

Pour voir le résultat de ce programme, supposez qu'un fichier `adcopystyle.txt` contienne le texte suivant :

```
The "SPACESHOT" shell hovers
at 100 meters for 2 to 3
minutes, erupting star bursts every 10 seconds that
generate ABUNDANT reading-level light for a
typical stadium.
```

Vous pourriez exécuter le programme `ShowFilters` à partir de la ligne de commande, comme ceci :

```
>ShowFilters adcopystyle.txt adout.txt
```

Le contenu du fichier `adout.txt` apparaîtrait ainsi :

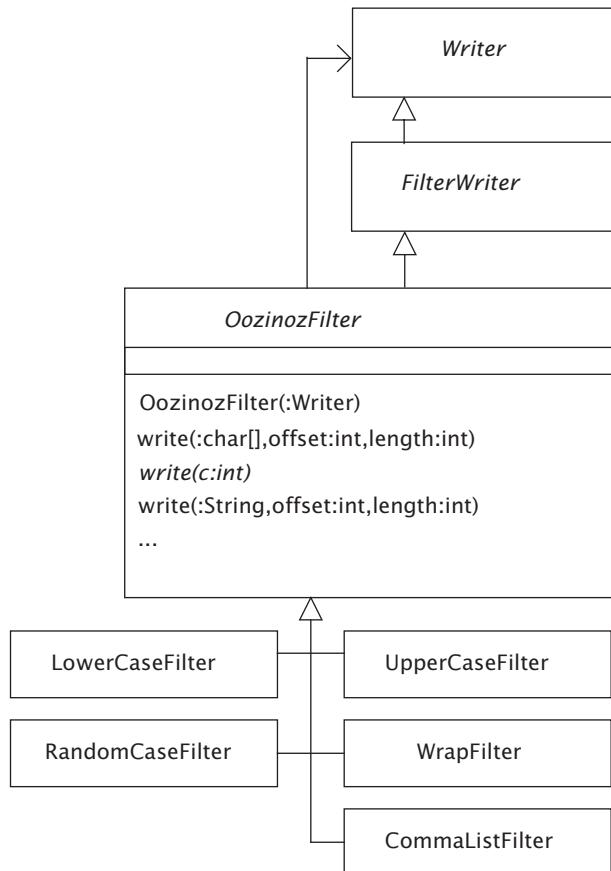
```
The "Spaceshot" Shell Hovers At 100
Meters For 2 To 3 Minutes, Erupting Star
```

Bursts Every 10 Seconds That Generate
Abundant Reading-level Light For A
Typical Stadium.

Plutôt que d'écrire dans un fichier, il peut être utile d'envoyer les caractères vers la console. La Figure 27.3 illustre la conception d'une classe qui étend Writer et dirige la sortie vers la console.

Figure 27.3

Un objet ConsoleWriter peut servir d'argument au constructeur de n'importe laquelle des sous-classes de OozinozFilter.



Exercice 27.2

Ecrivez le code de `ConsoleWriter.java`.

Les flux d'entrée et de sortie représentent un exemple classique de la façon dont le pattern DECORATOR permet d'assembler le comportement d'un objet au moment de l'exécution. Une autre application importante de ce pattern intervient lorsque vous avez besoin de créer des fonctions mathématiques à l'exécution.

Enveloppeurs de fonctions

Le principe de composer de nouveaux comportements lors de l'exécution au moyen du pattern DECORATOR s'applique aussi bien aux flux d'entrée/sortie qu'aux fonctions mathématiques. La possibilité de créer des fonctions lors de l'exécution est quelque chose dont vous pouvez faire profiter les utilisateurs, en leur permettant de spécifier de nouvelles fonctions *via* une interface GUI ou un petit langage. Vous pouvez aussi simplement vouloir réduire le nombre de méthodes présentes dans votre code et offrir davantage de souplesse en créant des fonctions mathématiques en tant qu'objets.

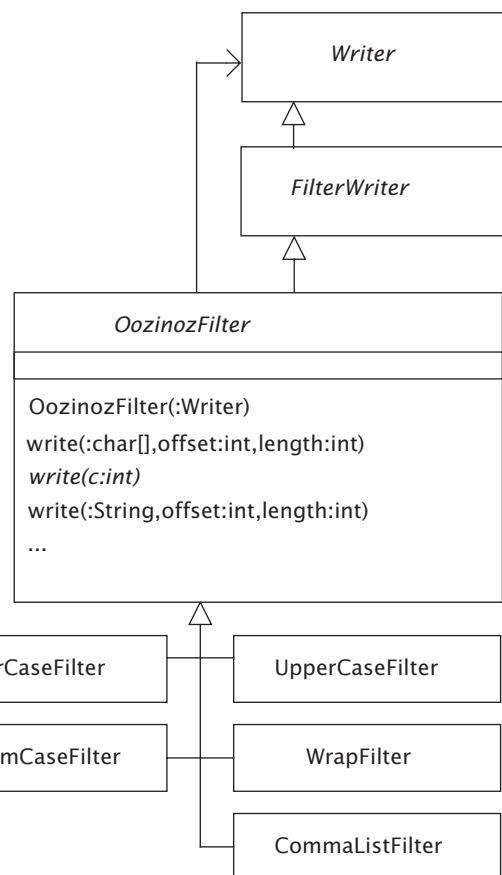
Pour créer une bibliothèque de décorateurs de fonctions, ou "enveloppeurs" de fonctions, nous pouvons appliquer une structure semblable à celle utilisée pour les flux d'entrée/sortie. Nous nommerons **Function** la super-classe enveloppeur. Pour la conception initiale de cette classe, nous pourrions copier la conception de la classe **OozinozFilter**, comme illustré Figure 27.4.

La classe **OozinozFilter** étend **FilterWriter** et son constructeur s'attend à recevoir un objet **Writer**. La conception de la classe **Function** est analogue, sauf qu'au lieu de recevoir un seul objet **IFunction**, elle accepte un tableau. Certaines fonctions, telles celles arithmétiques, nécessitent plus d'une fonction subordonnée. Dans le cas des enveloppeurs de fonctions, aucune classe existante telle que **Writer** n'implémente l'opération dont nous avons besoin. Nous pouvons donc nous passer d'une interface **IFunction** et définir plus simplement la hiérarchie **Function** sans cette interface, comme le montre la Figure 27.5.

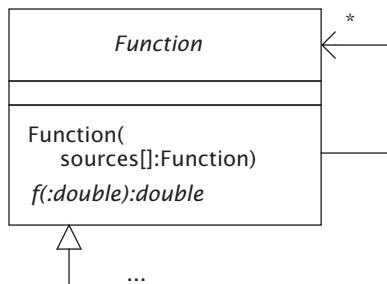
A l'instar de la classe **OozinozFilter**, la classe **Function** définit une opération commune que ses sous-classes doivent implémenter. Un choix naturel pour le nom de cette opération est **f**. Nous pourrions prévoir d'implémenter des fonctions paramétriques fondées sur un paramètre de temps qui varie de 0 à 1 (reportez-vous à l'encadré sur les équations paramétriques du Chapitre 4 pour une présentation du sujet).

Figure 27.4

La conception initiale de la hiérarchie d'enveloppeurs de fonctions ressemble beaucoup à la conception des flux d'entrée/sortie.

**Figure 27.5**

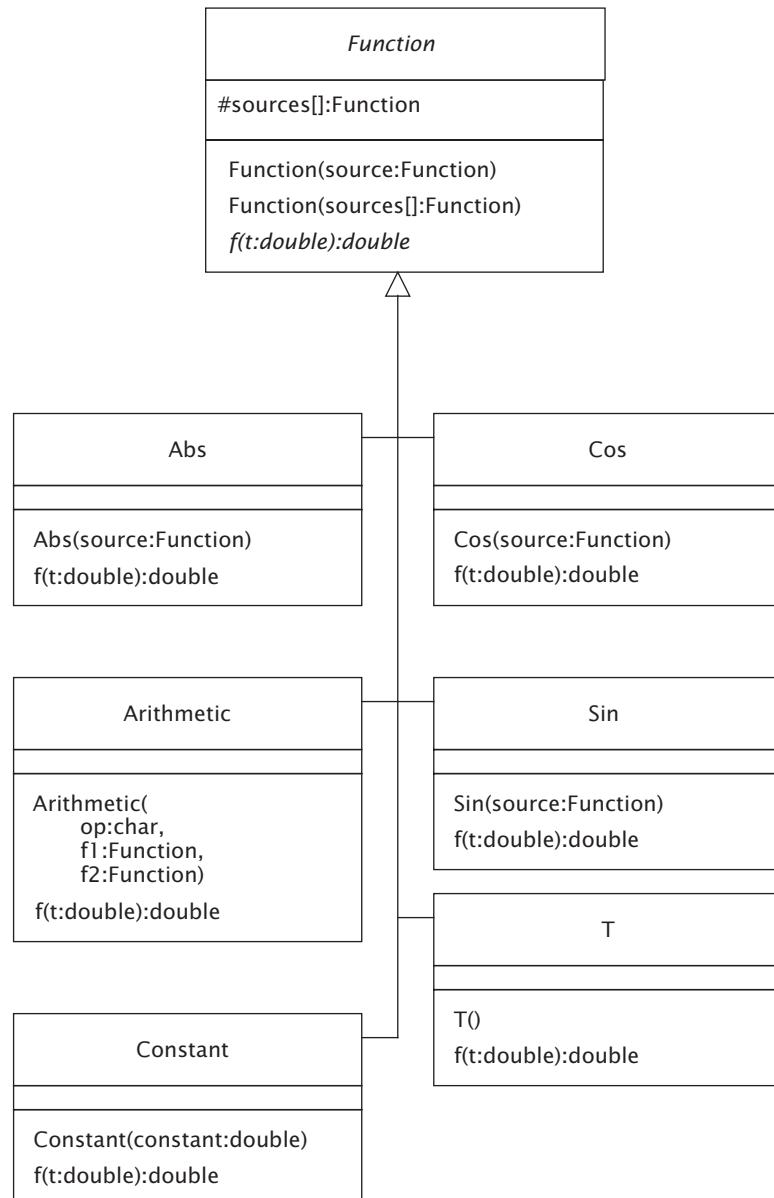
Une conception simplifiée pour la classe *Function* fonctionne sans définir d'interface séparée.



Nous allons créer une sous-classe de `Function` pour chaque enveloppeur. La Figure 27.6 présente une hiérarchie `Function` initiale.

Figure 27.6

Chaque sous-classe de `Function` implémente la fonction $f(t)$ de sorte qu'elle corresponde au nom de la classe.



Le code de la super-classe Function sert principalement à déclarer le tableau de sources :

```
package com.oozinoz.function;

public abstract class Function {
    protected Function[] sources;

    public Function(Function f) {
        this(new Function[] { f });
    }

    public Function(Function[] sources) {
        this.sources = sources;
    }

    public abstract double f(double t);

    public String toString() {
        String name = this.getClass().toString();
        StringBuffer buf = new StringBuffer(name);
        if (sources.length > 0) {
            buf.append('(');
            for (int i = 0; i < sources.length; i++) {
                if (i > 0)
                    buf.append(", ");
                buf.append(sources[i]);
            }
            buf.append(')');
        }
        return buf.toString();
    }
}
```

Les sous-classes de Function sont généralement simples. Voici par exemple le code de la classe Cos :

```
package com.oozinoz.function;
public class Cos extends Function {
    public Cos(Function f) {
        super(f);
    }

    public double f(double t) {
        return Math.cos(sources[0].f(t));
    }
}
```

Le constructeur de `Cos` attend un argument `Function` et le passe ensuite au constructeur de la super-classe, où il est stocké dans le tableau de sources. La méthode `Cos.f()` évalue la fonction source à l'heure `t`, passe cette valeur à la méthode `Math.Cos()` et retourne le résultat.

Les classes `Abs` et `Sin` sont presque identiques à `Cos`. La classe `Constant` vous permet de créer un objet `Function` contenant une valeur constante à retourner en réponse aux appels de la méthode `f()`. La classe `Arithmetic` accepte un indicateur d'opérateur qu'elle applique à sa méthode `f()`. Voici le code de cette classe :

```
package com.oozinoz.function;
public class Arithmetic extends Function {
    protected char op;

    public Arithmetic(char op, Function f1, Function f2) {
        super(new Function[] { f1, f2 });
        this.op = op;
    }

    public double f(double t) {
        switch (op) {
            case '+':
                return sources[0].f(t) + sources[1].f(t);
            case '-':
                return sources[0].f(t) - sources[1].f(t);
            case '*':
                return sources[0].f(t) * sources[1].f(t);
            case '/':
                return sources[0].f(t) / sources[1].f(t);
            default:
                return 0;
        }
    }
}
```

La classe `T` retourne les valeurs de `t` qui ont été passées. Ce comportement est utile si vous avez besoin d'une variable qui varie dans le temps de manière linéaire. Par exemple, l'expression suivante crée un objet `Function` dont la valeur de `f()` varie de 0 à 2π à mesure que le temps varie de 0 à 1 :

```
new Arithmetic('*', new T(), new Constant(2 * Math.PI))
```

Vous pouvez utiliser les classes `Function` pour composer de nouvelles fonctions mathématiques sans avoir à écrire de nouvelles méthodes. La classe `FunPanel` accepte des arguments `Function` pour ses fonctions `x` et `y`. Elle adapte aussi ces

fonctions à la taille du canevas. Cette classe peut être utilisée par un programme comme le suivant :

```
package app.decorator;

import app.decorator.brightness.FunPanel;

import com.oozinoz.function.*;
import com.oozinoz.ui.SwingFacade;

public class ShowFun {
    public static void main(String[] args) {
        Function theta = new Arithmetic(
            '*', new T(), new Constant(2 * Math.PI));
        Function theta2 = new Arithmetic(
            '*', new T(), new Constant(2 * Math.PI * 5));
        Function x = new Arithmetic(
            '+', new Cos(theta), new Cos(theta2));
        Function y = new Arithmetic(
            '+', new Sin(theta), new Sin(theta2));

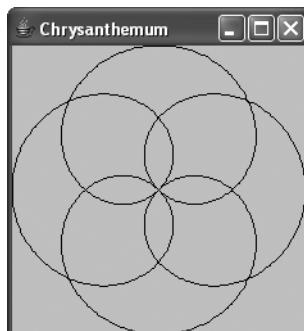
        FunPanel panel = new FunPanel(1000);
        panel.setPreferredSize(
            new java.awt.Dimension(200, 200));

        panel.setXY(x, y);
        SwingFacade.launch(panel, "Chrysanthemum");
    }
}
```

Ce programme utilise une fonction qui laisse un cercle s'entrelacer avec un autre plusieurs fois. Il produit le résultat de la Figure 27.7.

Figure 27.7

Une fonction mathématique complexe créée sans introduire aucune méthode nouvelle.



Pour étendre votre kit d'enveloppeurs de fonctions, il suffit d'ajouter de nouvelles fonctions mathématiques à la hiérarchie Function.

Exercice 27.3

Fermez ce livre et écrivez le code de la classe enveloppeur Exp.

Supposez que la luminosité d'une étoile soit une onde sinusoïdale qui décroît exponentiellement :

$$\text{luminosité} = e^{-4t} \cdot \sin(\pi t)$$

Comme précédemment, nous pouvons composer une fonction sans avoir à écrire de nouvelles classes ou méthodes :

```
package app.decorator.brightness;

import com.oozinoz.function.*;
import com.oozinoz.ui.SwingFacade;

public class ShowBrightness {
    public static void main(String args[]) {
        FunPanel panel = new FunPanel();
        panel.setPreferredSize(
            new java.awt.Dimension(200, 200));

        Function brightness = new Arithmetic(
            '*',
            new Exp(
                new Arithmetic(
                    '*',
                    new Constant(-4),
                    new T())),
            new Sin(
                new Arithmetic(
                    '*',
                    new Constant(Math.PI),
                    new T())));

        panel.setXY(new T(), brightness);
        SwingFacade.launch(panel, "Brightness");
    }
}
```

Ce code produit la courbe de la Figure 27.8.

Figure 27.8

La luminosité d'une étoile connaît un pic soudain avant de décroître.



Exercice 27.4

Ecrivez le code pour définir un objet `Brightness` représentant la fonction de luminosité.

A mesure que vous en avez besoin, vous pouvez ajouter d'autres fonctions à la hiérarchie `Function`. Par exemple, des classes pour la racine carrée et la tangente pourraient être utiles. Vous pouvez aussi créer de nouvelles hiérarchies applicables à différents types, tels que des chaînes, ou impliquant une définition différente de l'opération `f()`. Par exemple, `f()` pourrait être définie en tant qu'une fonction de temps à deux ou trois dimensions. Indépendamment de la hiérarchie que vous créez, vous pouvez utiliser le pattern `DECORATOR` pour développer un riche ensemble de fonctions composable lors de l'exécution.

DECORATOR en relation avec d'autres patterns

Le pattern `DECORATOR` s'appuie sur une opération commune implémentée à travers une hiérarchie. A cet égard, il ressemble à `STATE`, `STRATEGY` et `INTERPRETER`. Dans `DECORATOR`, les classes possèdent aussi habituellement un constructeur qui requiert un autre objet décorateur subordonné. Ce pattern ressemble sur ce point à `COMPOSITE`. `DECORATOR` ressemble également à `PROXY` en ce que les classes décorateur implémentent typiquement l'opération commune en transmettant l'appel à l'objet décorateur subordonné.

Résumé

Le pattern DECORATOR permet d'assembler des variations d'une même opération. Un exemple classique apparaît dans les flux d'entrée/sortie, où vous pouvez composer un flux à partir d'un autre flux. Les bibliothèques de classes Java supportent DECORATOR dans l'implémentation des flux d'entrée/sortie. Vous pouvez étendre ce principe en créant votre propre ensemble de filtres d'entrée/sortie. Vous pouvez également appliquer DECORATOR pour définir des enveloppeurs de fonctions permettant de créer un large ensemble d'objets fonction à partir d'un jeu fixe de classes de fonctions. Ce pattern donne lieu à des conceptions flexibles dans les situations où vous voulez pouvoir combiner les variations d'implémentation d'une opération commune en de nouvelles variations au moment de l'exécution.

ITERATOR

Etendre les fonctionnalités d'un code existant en ajoutant un nouveau type de collection peut requérir l'ajout d'un itérateur. Ce chapitre étudie le cas particulier de l'itération parcourant un objet composite. Outre l'itération dans de nouveaux types de collections, le cas d'un environnement multithread soulève un certain nombre de problèmes intéressants qui méritent d'être analysés. Simple de prime abord, l'itération est en fait un problème non totalement résolu.

L'objectif du pattern ITERATOR est de fournir un moyen d'accéder de façon séquentielle aux éléments d'une collection.

Itération ordinaire

Java dispose de différentes approches pour réaliser une itération :

- les boucles `for`, `while` et `repeat`, se fondant généralement sur un indice ;
- la classe `Enumeration` (dans `java.util`) ;
- la classe `Iterator` (également dans `java.util`), ajoutée dans le JDK 1.2 pour gérer les collections ;
- les boucles `for` étendues (`foreach`), ajoutées dans le JDK 1.5.

Nous utiliserons la classe `Iterator` pour une grande partie de ce chapitre, cette section présentant une boucle `for` étendue.

Une classe `Iterator` possède trois méthodes : `hasnext()`, `next()` et `remove()`. Un itérateur a le droit de générer une exception du type `UnsupportedOperationException` s'il ne supporte pas l'opération `remove()`.

Une boucle `for` étendue suit la syntaxe suivante :

```
for (Type element : collection)
```

Cette instruction crée une boucle lisant une collection, extrayant un élément à la fois (appelé ici `element`). Il n'est pas nécessaire de convertir `element` en un type particulier, cela étant géré de façon implicite. Cette construction peut aussi fonctionner avec des tableaux (array). Une classe qui souhaite autoriser des boucles `for` étendues doit implémenter une interface `Iterable` et inclure une méthode `iterator()`.

Voici un programme illustrant la classe `Iterator` et des boucles `for` étendues :

```
package app.iterator;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class ShowForeach {
    public static void main(String[] args) {
        ShowForeach example = new ShowForeach();
        example.showIterator();
        System.out.println();
        example.showForeach();
    }

    public void showIterator() {
        List names = new ArrayList();
        names.add("Fuser:1101");
        names.add("StarPress:991");
        names.add("Robot:1");

        System.out.println("Itérateur style JDK 1.2 :");
        for (Iterator it = names.iterator(); it.hasNext();) {
            String name = (String) it.next();
            System.out.println(name);
        }
    }

    public void showForeach() {
        List<String> names = new ArrayList<String>();
        names.add("Fuser:1101");
        names.add("StarPress:991");
        names.add("Robot:1");

        System.out.println(
            "Boucle FOR étendue style JDK 1.5 :");
        for (String name: names)
```

```
        System.out.println(name);
    }
}
```

Lorsque nous exécutons ce programme, nous voyons les résultats :

```
Itérateur style JDK 1.2 :
Fuser:1101
StarPress:991
Robot:1

Boucle FOR étendue style JDK 1.5 :
Fuser:1101
StarPress:991
Robot:1
```

Pour l'instant, la société Oozinoz doit continuer à utiliser l'ancien style de classes `Iterator`. Elle ne peut augmenter la version du code avant d'être absolument sûre que ses clients disposent des nouveaux compilateurs. Vous pouvez néanmoins essayer aujourd'hui les boucles `for` génériques et étendues.

Itération avec sécurité inter-threads

Les applications riches en fonctionnalités emploient souvent des threads pour réaliser des tâches s'exécutant avec une apparence de simultanéité. En particulier, il est fréquent d'exécuter en arrière-plan les tâches coûteuses en temps pour ne pas ralentir la réactivité de la GUI. L'utilisation de threads est utile, mais elle comporte ses risques. De nombreuses applications plantent en raison de tâches s'exécutant dans des threads qui ne collaborent pas efficacement. Des méthodes parcourant une collection peuvent par exemple être en cause.

Les classes de collection dans `java.util.Collections` offrent une certaine mesure de sécurité de thread en fournissant une méthode `synchronized()`. En essence, elle retourne une version de la collection sous-jacente, évitant ainsi que deux threads la modifient en même temps.

Une collection et son itérateur coopèrent pour détecter si une liste change durant l'itération, c'est-à-dire si la liste est synchronisée. Pour observer ce comportement en action, supposez que le singleton Factory d'Oozinoz puisse nous indiquer les machines qui sont actives à un certain moment et que nous souhaitions en afficher la liste. L'exemple de code dans le package `app.iterator.concurrent` implémente cette liste dans la méthode `upMachineNames()`.

Le programme suivant affiche une liste des machines qui sont actuellement actives, mais simule la condition que de nouvelles machines puissent entrer en action alors que le programme est en train d'afficher la liste :

```
package app.iterator.concurrent;
import java.util.*;

public class ShowConcurrentIterator implements Runnable {
    private List list;

    protected static List upMachineNames() {
        return new ArrayList(Arrays.asList(new String[] {
            "Mixer1201", "ShellAssembler1301",
            "StarPress1401", "UnloadBuffer1501" }));
    }

    public static void main(String[] args) {
        new ShowConcurrentIterator().go();
    }

    protected void go() {
        list = Collections.synchronizedList(
            upMachineNames());
        Iterator iter = list.iterator();
        int i = 0;
        while (iter.hasNext()) {
            i++;
            if (i == 2) { // simule l'activation d'une machine
                new Thread(this).start();
                try { Thread.sleep(100); }
                catch (InterruptedException ignored) {}
            }
            System.out.println(iter.next());
        }
    }
    /**
     * Insère un élément dans la liste, dans un thread distinct.
     */
    public void run() {
        list.add(0, "Fuser1101");
    }
}
```

La méthode `main()` dans ce code construit une instance de la classe et appelle la méthode `go()`. Cette méthode parcourt par itération la liste des machines actives, en prenant soin d'en construire une version synchronisée. Ce code simule la situation où une nouvelle machine devient active alors que la méthode lit la liste. La méthode `run()` modifie la liste, s'exécutant dans un thread séparé.

Le programme ShowConcurrentIterator affiche une ou deux machines puis plante :

```
Mixer1201
java.util.ConcurrentModificationException
at java.util.AbstractList$Itr.checkForComodification(Unknown Source)
at java.util.AbstractList$Itr.next(Unknown Source)
at
com.oozinoz.app.ShowConcurrent>ShowConcurrentIterator.go(ShowConcurrent
Iterator.java:49)
at
com.oozinoz.app.ShowConcurrent>ShowConcurrentIterator.main(ShowConcurre
ntIterator.java:29)
Exception in thread "main" .
```

Le programme plante car la liste et les objets itérateurs détectent que la liste a changé durant l'itération. Vous n'avez pas besoin de créer un nouveau thread pour illustrer ce comportement. Vous pouvez écrire un programme qui plante simplement en modifiant une collection à partir d'une boucle d'énumération. Dans la pratique, c'est plus vraisemblablement par accident qu'une application multithread modifie une liste pendant qu'un itérateur la parcourt.

Nous *pouvons* élaborer une approche par thread sécurisée pour lire une liste. Toutefois, il est important de noter que le programme ShowConcurrentIterator plante seulement parce qu'il utilise un itérateur. L'énumération par boucle `for` des éléments d'une liste, même synchronisée, ne déclenchera pas l'exception démontrée dans le programme précédent, mais elle pourra néanmoins rencontrer des problèmes. Considérez cette version du programme :

```
package app.iterator.concurrent;
import java.util.*;

public class ShowConcurrentFor implements Runnable {
    private List list;

    protected static List upMachineNames() {
        return new ArrayList(Arrays.asList(new String[] {
            "Mixer1201", "ShellAssembler1301",
            "StarPress1401", "UnloadBuffer1501" }));
    }

    public static void main(String[] args) {
        new ShowConcurrentFor().go();
    }

    protected void go() {
        System.out.println(
            "Cette version permet l'ajout concurrent"
            + " de nouveaux éléments :");
```

```
list = Collections.synchronizedList(
    upMachineNames());
display();
}

private void display() {
    for (int i = 0; i < list.size(); i++) {
        if (i == 1) { // simule l'activation d'une machine
            new Thread(this).start();
            try { Thread.sleep(100); }
            catch (InterruptedException ignored) {}
        }
        System.out.println(list.get(i));
    }
}
/**
 * Insère un élément dans la liste, dans un thread distinct.
 */
public void run() {
    list.add(0, "Fuser1101");
}
```

L'exécution de ce programme affiche :

```
Cette version permet l'ajout concurrent de nouveaux éléments :
Mixer1201
Mixer1201
ShellAssembler1301
StarPress1401
UnloadBuffer1501
```

Exercice 28.1

Expliquez le résultat en sortie du programme `ShowConcurrentFor`.

- *Les solutions des exercices de ce chapitre sont données dans l'Annexe B.*

Nous avons examiné deux versions de l'exemple de programme : une qui plante et une qui produit une sortie incorrecte. Aucun de ces résultats n'étant acceptable, nous devons recourir à une autre approche pour protéger une liste pendant l'énumération de ses éléments.

Il y a deux approches courantes pour coder une itération sur une collection dans une application multithread. Elles impliquent toutes deux l'emploi d'un objet appelé

mutex, qui est partagé par des threads rivalisant pour obtenir le contrôle du verrou sur l'objet. La première possibilité de conception est de forcer tous les threads à obtenir le contrôle du verrou du mutex avant de pouvoir accéder à la collection. Voici un exemple de cette approche :

```
package app.iterator.concurrent;
import java.util.*;

public class ShowConcurrentMutex implements Runnable {
    private List list;

    protected static List upMachineNames() {
        return new ArrayList(Arrays.asList(new String[] {
            "Mixer1201", "ShellAssembler1301",
            "StarPress1401", "UnloadBuffer1501" }));
    }

    public static void main(String[] args) {
        new ShowConcurrentMutex().go();
    }

    protected void go() {
        System.out.println(
            "Cette version synchronise correctement :");
        list = Collections.synchronizedList(upMachineNames());
        synchronized (list) {
            display();
        }
    }

    private void display() {
        for (int i = 0; i < list.size(); i++) {
            if (i == 1) { // simule l'activation d'une machine
                new Thread(this).start();
                try { Thread.sleep(100);
                } catch (InterruptedException ignored) {}
            }
            System.out.println(list.get(i));
        }
    }
    /**
     * Insère un élément dans la liste, dans un thread distinct.
     */
    public void run() {
        synchronized (list) {
            list.add(0, "Fuser1101");
        }
    }
}
```

Ce programme affiche la liste originale :

```
Cette version synchronise correctement :  
Mixer1201  
ShellAssembler1301  
StarPress1401  
UnloadBuffer1501
```

La sortie du programme montre la liste telle qu'elle existait avant l'insertion d'un nouvel objet par la méthode `run()`. Le programme retourne un résultat cohérent, sans duplication, car la logique du programme requiert que la méthode `run()` attende la fin de l'énumération des éléments dans la méthode `display()`. Bien que les résultats soient corrects, la conception peut se révéler impossible à implémenter. En effet, il est probable que vous n'ayez pas le luxe d'avoir des threads bloqués pendant qu'un thread exécute son itération.

La deuxième solution consiste à cloner la collection dans une opération avec un mutex puis de lire le contenu du clone. L'avantage du clonage d'une liste avant de la parcourir est la vitesse. C'est une opération souvent plus rapide que d'attendre qu'une méthode finisse son travail sur le contenu d'une collection. Cette approche peut toutefois donner lieu à des problèmes.

La méthode `clone()` pour `ArrayList` produit une copie "partielle", c'est-à-dire une nouvelle collection qui se réfère aux mêmes objets que la collection originale. Cette approche échouerait donc si d'autres threads pouvaient modifier les objets sous-jacents d'une manière qui interfère avec votre méthode. Dans certains cas toutefois, ce risque est faible. Par exemple, si vous souhaitez simplement afficher une liste de noms de machines, il est peu probable que des noms changent au moment où votre méthode lit le clone de la liste.

Pour résumer, nous avons vu quatre approches différentes permettant d'énumérer par itération les éléments d'une liste dans un environnement multithread. Deux d'entre elles emploient la méthode `synchronized()` et sont fautives, pouvant soit planter soit produire des résultats incorrects. Les deux dernières que nous avons étudiées emploient le verrouillage et le clonage pour produire des résultats corrects, mais elles ont aussi leurs inconvénients.

Java et ses bibliothèques fournissent un support substantiel pour l'itération dans un environnement multithread, mais ce support ne vous affranchit pas des difficultés de la conception avec des processus concurrents. Les bibliothèques Java offrent de

bonnes fonctionnalités pour gérer la lecture des nombreuses collections qu'elles proposent, mais si vous introduisez votre propre type de collection, il vous faudra aussi introduire un itérateur qui lui soit associé.

Exercice 28.2

Donnez un argument contre l'emploi de la méthode `synchronized()`, ou justifiez le fait qu'une approche par verrouillage n'est pas non plus toujours la réponse.

Itération sur un objet composite

Il est généralement facile de concevoir des algorithmes qui parcouruent une structure composite, visitant chaque nœud et réalisant certaines tâches. L'Exercice 5.3 du Chapitre 5 vous avait demandé de concevoir plusieurs algorithmes s'exécutant de manière récursive pour parcourir une structure composite. La création d'un itérateur peut être beaucoup plus complexe que la conception d'un algorithme récursif. La difficulté réside dans le transfert en retour du contrôle à une autre partie du programme et la conservation d'un genre de signet permettant à l'itérateur de reprendre là où il avait suspendu son travail.

Les composites fournissent un bon exemple d'itérateur difficile à développer.

Vous pouvez penser que vous aurez besoin d'une nouvelle classe d'itérateur pour chaque composite spécifique que vous créerez. Vous pouvez en fait concevoir un itérateur composite relativement réutilisable, hors le fait qu'il vous faudra modifier vos classes de composite pour retourner le bon type d'itérateur.

La conception d'un itérateur composite est aussi naturellement récursive que les composites le sont eux-mêmes. Pour lire par itération un composite, il faut lire ses enfants, bien que ce soit un peu plus complexe que cela puisse paraître de prime abord. Nous avons le choix entre retourner un nœud avant ou après ses descendants (techniques appelées respectivement *traversée pré-ordonnée* ou *post-ordonnée*). Si nous choisissons une traversée pré-ordonnée, nous devons énumérer les enfants après avoir retourné la tête, en prenant soin de noter que chaque enfant peut lui-même être un composite. Une subtilité demande ici que nous gérions deux itérateurs. Un itérateur, nommé 1 dans la Figure 28.1, garde trace de son enfant actuel.

C'est un simple itérateur de liste qui parcourt la liste d'enfants. Un second itérateur (nommé 2) parcourt l'enfant actuel en tant que composite. La Figure 28.1 illustre les trois aspects de la détermination du nœud actuel dans une itération sur un composite.

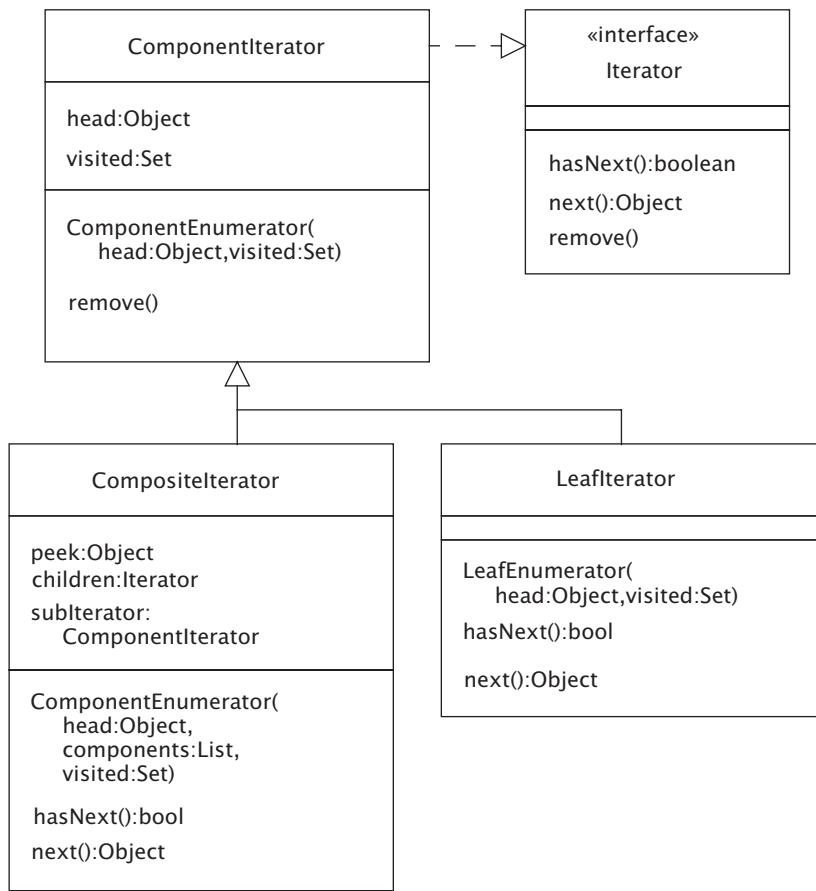
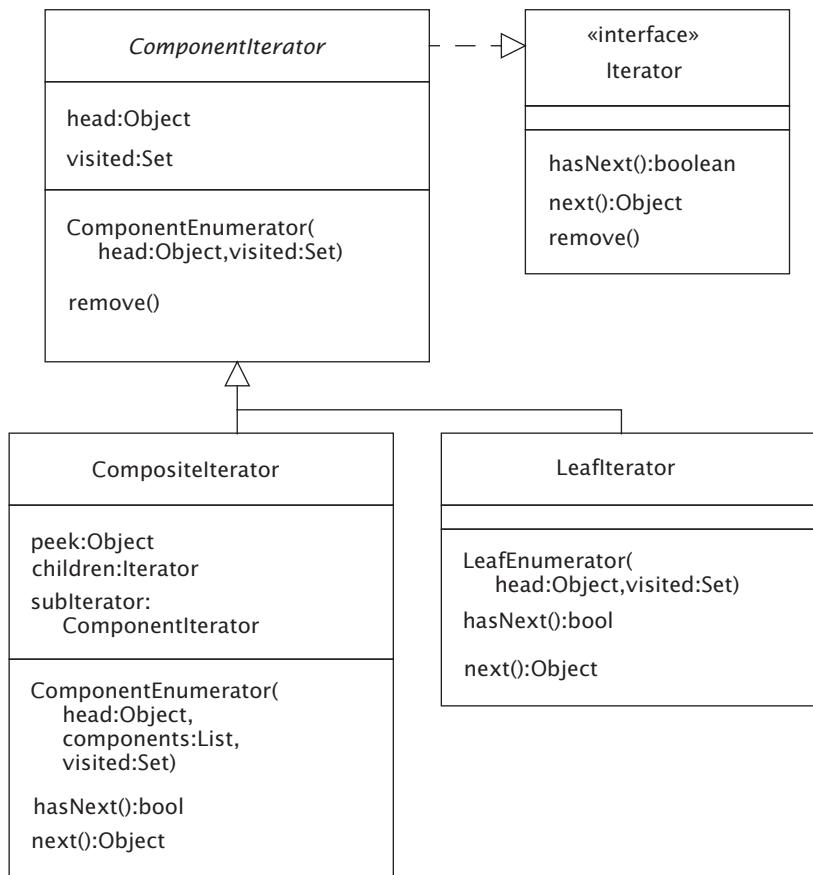


Figure 28.1

La lecture par itération d'un composite nécessite : de signaler la tête(); de parcourir séquentiellement les enfants (1), et de parcourir un enfant composite.

Pour notre travail de conception d'un itérateur composite, nous pouvons supposer qu'une itération sur un nœud simple sera une chose banale alors qu'une itération sur un nœud composite sera plus difficile. En première approximation, nous pouvons imaginer une conception d'itérateurs ressemblant à l'exemple illustré Figure 28.2.

**Figure 28.2**

Une première conception pour une famille d'énumérateurs par itération.

Les classes emploient les noms de méthodes `hasNext()` et `next()` pour que la classe `ComponentIterator` implémente l'interface `Iterator` du package `java.util`.

La conception montre que les constructeurs de classe d'énumération acceptent un objet à parcourir par itération. Dans la pratique, cet objet sera un composite, de machines ou de processus, par exemple. La conception utilise aussi une variable `visited` pour garder trace des nœuds déjà énumérés. Cela nous évite d'entrer dans

une boucle infinie lorsqu'un composite a des cycles. Le code pour ComponentIterator au sommet de la hiérarchie aura finalement l'apparence suivante :

```
package com.oozinoz.iterator;
import java.util.*;

public abstract class ComponentIterator
    implements Iterator {
    protected Object head;
    protected Set visited;
    protected boolean returnInterior = true;

    public ComponentIterator(Object head, Set visited) {
        this.head = head;
        this.visited = visited;
    }

    public void remove() {
        throw new UnsupportedOperationException(
            "ComponentIterator.Remove");
    }
}
```

Cette classe laisse la plus grande part du travail difficile à ses sous-classes.

Dans la sous-classe CompositeIterator, nous pouvons anticiper le besoin pour un itérateur de liste d'énumérer les enfants d'un nœud composite. C'est l'énumération notée 1 dans la Figure 28.1, représentée par la variable `children` dans la Figure 28.2. Les composites ont également besoin d'un énumérateur pour l'énumération notée 2 dans la figure. La variable `subIterator` dans la Figure 28.2 répond à ce besoin. Le constructeur de la classe CompositeEnumerator peut initialiser l'énumérateur d'enfants de la manière suivante :

```
public CompositeIterator(
    Object head, List components, Set visited) {
    super(head, visited);
    children = components.iterator();
}
```

Lorsque nous commençons la traversée d'un composite, nous savons que le premier nœud à retourner est le nœud de tête (marqué H dans la Figure 28.1). Ainsi, le code pour la méthode `next()` d'une classe CompositeIterator pourrait être comme suit :

```
public Object next() {
    if (peek != null) {
        Object result = peek;
```

```

        peek = null;
        return result;
    }

    if (!visited.contains(head)) {
        visited.add(head);
        return head;
    }

    return nextDescendant();
}

```

La méthode `next()` utilise l'ensemble visité (`Set visited`) pour enregistrer si l'énumérateur a déjà retourné le nœud de tête. S'il a retourné la tête d'un composite, la méthode `nextDescendant()` doit trouver le nœud suivant.

A tout moment, la variable `subIterator` peut se trouver en cours de processus d'énumération d'un enfant qui est lui-même un nœud composite. Si cet énumérateur est actif, la méthode `next()` de la classe `CompositeIterator` peut "déplacer" le sous-itérateur. Si celui-ci ne peut se déplacer, le code doit mettre le prochain élément dans la liste d'enfants, obtenir un nouveau sous-itérateur pour lui, et déplacer cet énumérateur. Le code de la méthode `nextDescendant()` montre cette logique :

```

protected Object nextDescendant() {
    while (true) {
        if (subiterator != null) {
            if (subiterator.hasNext())
                return subiterator.next();
        }

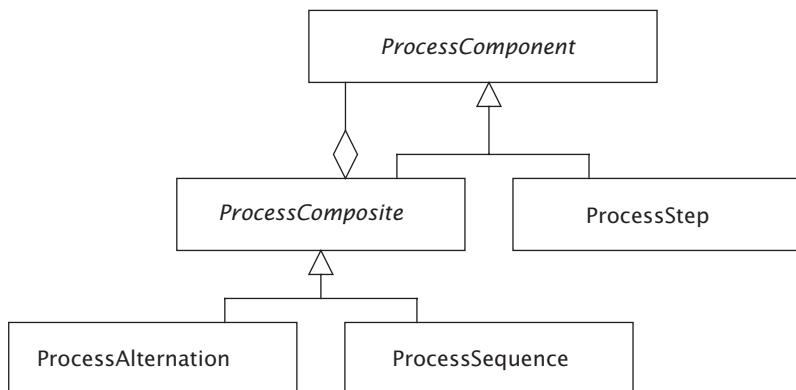
        if (!children.hasNext()) return null;
        ProcessComponent pc =
            (ProcessComponent) children.next();
        if (!visited.contains(pc)) {
            subiterator = pc.iterator(visited);
        }
    }
}

```

Cette méthode introduit la première contrainte que nous avons rencontrée concernant le type d'objets que nous pouvons énumérer : le code requiert que les enfants dans un composite implémentent une méthode `iterator(:Set)`. Considérez un exemple d'une structure composite, telle que la hiérarchie `ProcessComponent` que le Chapitre 5 a introduite. La Figure 28.3 illustre la hiérarchie de composites de processus qu'Oozinoz utilise pour modéliser le flux de processus de fabrication qui produit les divers types d'artifices.

Figure 28.3

Le flux des processus de fabrication d'Oozinoz est constitué de composites.



La méthode `next()` de la classe `CompositeIterator` doit énumérer les nœuds de chaque enfant qui appartient à un objet composite. Nous devons faire en sorte que la classe de l'enfant implémente une méthode `iterator(:Set)` que le code de `next()` puisse utiliser. La Figure 28.2 illustre la relation qui unit les classes et les interfaces.

Pour actualiser la hiérarchie de `ProcessComponent` afin de pouvoir la parcourir, nous devons prévoir une méthode `iterator()` :

```

public ComponentIterator iterator() {
    return iterator(new HashSet());
}
public abstract ComponentIterator iterator(Set visited);
  
```

La classe `ProcessComponent` est abstraite et la méthode `iterator(:Set)` doit être implémentée par les sous-classes. Pour la classe `ProcessComposite`, le code se présenterait comme suit :

```

public ComponentIterator iterator(Set visited) {
    return new CompositeIterator(this, subprocesses, visited);
}
  
```

Voici l'implémentation de `iterator()` dans la classe `ProcessStep` :

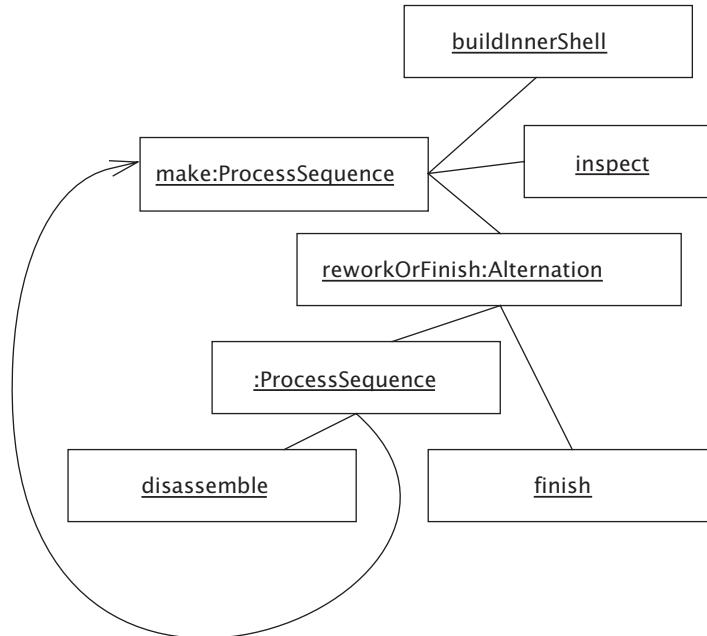
```

public ComponentIterator iterator(Set visited) {
    return new LeafIterator(this, visited);
}
  
```

Avec ces petits ajouts en place dans la hiérarchie `ProcessComponent`, nous pouvons maintenant écrire du code pour énumérer les nœuds d'un composite de processus.

Exercice 28.3

Quel pattern appliquez-vous pour permettre aux classes d'une hiérarchie `ProcessComponent` d'implémenter `iterator()` afin de créer des instances d'une classe d'itérateur appropriée ?

**Figure 28.4**

Le flux de processus de fabrication de bombes aériennes est un composite cyclique. Chaque nœud feuille dans ce diagramme est une instance de `ProcessStep`. Les autres nœuds sont des instances de `ProcessComposite`.

La Figure 28.4 illustre le modèle objet d'un flux de processus typique chez Oozinoz. Le court programme qui suit énumère tous les nœuds de ce modèle :

```

package app.iterator.process;

import com.oozinoz.iterator.ComponentIterator;
import com.oozinoz.process.ProcessComponent;
import com.oozinoz.process.ShellProcess;

public class ShowProcessIteration {
    
```

```

public static void main(String[] args) {
    ProcessComponent pc = ShellProcess.make();
    ComponentIterator iter = pc.iterator();
    while (iter.hasNext())
        System.out.println(iter.next());
}
}

```

L'exécution de ce programme affiche les informations suivantes :

```

Fabriquer une bombe aérienne
Créer la coque interne
Contrôler
Retraiter la coque interne, ou finir la bombe
Retraiter
Désassembler
Terminer : ajouter la charge de propulsion, insérer le dispositif
d'allumage,
et emballer

```

Les étapes énumérées sont celles définies par la classe `ShellProcess`. Notez que, dans le modèle objet, l'étape qui suit "Désassembler" est "Fabriquer". Le résultat en sortie omet cela car l'énumération voit qu'elle a déjà énuméré cette étape dans la première ligne du résultat.

Ajout d'un niveau de profondeur à un énumérateur

La sortie de ce programme pourrait être plus claire si nous prévoyions chaque étape conformément à sa profondeur dans le modèle. Nous pouvons définir la profondeur d'un énumérateur de feuille comme étant 0 et celle d'un énumérateur de composite comme étant 1 plus la profondeur de son sous-itérateur. Nous pouvons déclarer `getDepth()` abstraite dans la super-classe `ComponentIterator` de la manière suivante :

```
public abstract int getDepth();
```

Le code pour la méthode `getDepth()` dans la classe `LeafIterator` se présente comme suit :

```
public int getDepth() {
    return 0;
}
```

Le code pour `CompositeIterator.getDepth()` est :

```
public int getDepth() {
    if (subiterator != null)
        return subiterator.getDepth() + 1;
    return 0;
}
```

Le programme suivant produit une sortie plus lisible :

```
package app.iterator.process;

import com.oozinoz.iterator.ComponentIterator;
import com.oozinoz.process.ProcessComponent;
import com.oozinoz.process.ShellProcess;

public class ShowProcessIteration2 {
    public static void main(String[] args) {
        ProcessComponent pc = ShellProcess.make();
        ComponentIterator iter = pc.iterator();
        while (iter.hasNext()) {
            for (int i = 0; i < 4 * iter.getDepth(); i++)
                System.out.print(' ');
            System.out.println(iter.next());
        }
    }
}
```

La sortie du programme est :

```
Fabriquer une bombe aérienne
Créer la coque interne
Contrôler
Retraiter la coque interne, ou finir la bombe
    Retraiter
    Désassembler
Terminer : ajouter la charge de propulsion, insérer le dispositif
    ↗ d'allumage, et emballer
```

Une autre amélioration que nous pouvons apporter à la hiérarchie `ComponentIterator` est de ne permettre que l'énumération des feuilles d'un composite.

Enumération des feuilles

Supposez que nous voulions permettre à une énumération de ne retourner que les feuilles. Cela pourrait être utile si nous étions intéressés par des attributs ne s'appliquant qu'à cette catégorie de nœuds, tel le temps requis par une étape procédurale. Nous pourrions ajouter un champ `returnInterior` à la classe `ComponentIterator` afin d'enregistrer si les nœuds intérieurs (non feuilles) devraient ou non être retournés par l'énumération :

```
protected boolean returnInterior = true;

public boolean shouldShowInterior() {
    return returnInterior;
}

public void setShowInterior(boolean value) {
    returnInterior = value;
}
```

Dans la méthode `nextDescendant()` de la classe `CompositeIterator`, nous aurons besoin de retransmettre cet attribut lorsque nous créerons une nouvelle énumération pour un enfant de nœud composite :

```
protected Object nextDescendant() {
    while (true) {
        if (subiterator != null) {
            if (subiterator.hasNext())
                return subiterator.next();
        }
        if (!children.hasNext()) return null;
        ProcessComponent pc =
            (ProcessComponent) children.next();
        if (!visited.contains(pc)) {
            subiterator = pc.iterator(visited);
            subiterator.setShowInterior(
                shouldShowInterior());
        }
    }
}
```

Il nous faudra aussi modifier la méthode `next()` de la classe `CompositeEnumerator`. Le code actuel est le suivant :

```
public Object next() {
    if (peek != null) {
        Object result = peek;
        peek = null;
        return result;
    }

    if (!visited.contains(head)) {
        visited.add(head);
    }

    return nextDescendant();
}
```

Exercice 28.4

Modifiez la méthode `next()` de la classe `CompositeIterator` pour respecter la valeur du champ `returnInterior`.

La création d'un itérateur, ou d'un énumérateur, pour un nouveau type de collection peut représenter une certaine somme de travail. L'avantage qui en résultera sera qu'il sera aussi simple de travailler avec votre collection qu'avec les bibliothèques de classes Java.

Résumé

L'objectif du pattern **ITERATOR** est de permettre à un client d'accéder en séquence aux éléments d'une collection. Les classes de collection dans les bibliothèques Java offrent un support avancé pour travailler avec des collections, dont la gestion d'itérations, ou l'énumération. Lorsque l'on crée un nouveau type de collection, on lui associe souvent un itérateur. Un composite spécifique à un domaine est un exemple courant de nouveau type de collection. Le support de la boucle `for`, générique comme étendu, améliorera la visibilité de votre code. Vous pouvez concevoir un itérateur relativement générique que vous pourrez ensuite appliquer à une variété de hiérarchies de composites.

Lorsque vous instanciez un itérateur, vous devez vous demander si la collection peut changer pendant que vous en énumérez les éléments. Dans une application mono-thread, il y a peu de chances que cela se produise, mais dans une application multithread, vous devrez vous assurer que l'accès à une collection est bien synchronisé. Pour énumérer des éléments en toute sécurité dans un tel environnement, vous pouvez synchroniser l'accès aux collections en verrouillant un objet mutex. Vous pouvez bloquer tous les accès au cours de l'énumération, ou brièvement pendant le clonage d'une collection. Avec une conception correcte, vous pouvez assurer une sécurité inter-threads aux clients de votre code d'itérateur.

VISITOR

Pour étendre une hiérarchie de classes, normalement vous ajoutez simplement des méthodes qui fournissent le comportement souhaité. Il peut arriver néanmoins que ce comportement ne soit pas cohérent avec la logique du modèle objet existant. Il se peut aussi que vous n'ayez pas accès au code existant. Dans de telles situations, il peut être impossible d'étendre le comportement de la hiérarchie sans modifier ses classes. Le pattern VISITOR permet justement au développeur d'une hiérarchie d'intégrer un support pour les cas où d'autres développeurs voudraient étendre son comportement.

A l'instar de INTERPRETER, VISITOR est le plus souvent placé au-dessus de COMPOSITE. Vous pourriez donc vouloir réviser COMPOSITE car nous nous y référerons dans ce chapitre.

L'objectif du pattern VISITOR est de vous permettre de définir une nouvelle opération pour une hiérarchie sans changer ses classes.

Application de VISITOR

Le pattern VISITOR permet, avec un peu de prévoyance lors du développement d'une hiérarchie de classes, d'ouvrir la voie à une variété illimitée d'extensions pouvant être apportées par un développeur n'ayant pas accès au code source. Voici comment l'appliquer :

- Ajoutez une opération `accept()` à certaines ou à toutes les classes d'une hiérarchie. Chaque implémentation de cette méthode acceptera un argument dont le type sera une interface que vous créerez.

- Créez une interface avec un ensemble d'opérations partageant un nom commun, habituellement `visit`, mais possédant des types d'arguments différents. Déclarez une de ces opérations pour chaque classe de la hiérarchie dont vous autorisez l'extension.

La Figure 29.1 illustre le diagramme de classes de la hiérarchie `MachineComponent` modifiée pour supporter `VISITOR`.

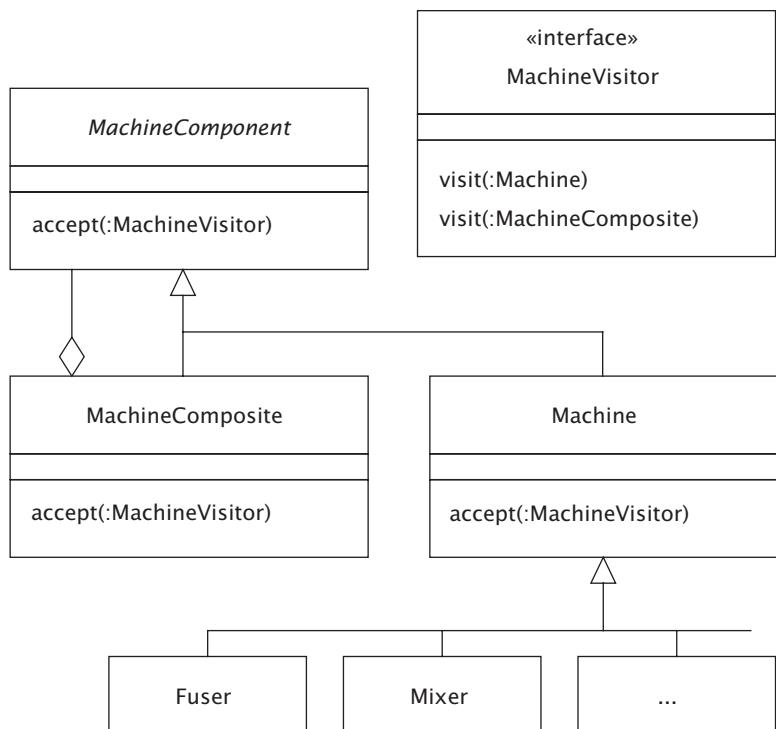


Figure 29.1

Pour intégrer le support de `VISITOR` à la hiérarchie `MachineComponent`, ajoutez les méthodes `accept()` et l'interface `MachineVisitor` présentées dans ce diagramme.

Ce diagramme n'explique pas comment `VISITOR` fonctionne, ce qui est l'objet de la prochaine section. Il montre simplement certains des principes vous permettant d'appliquer ce pattern.

Notez que toutes les classes du diagramme de MachineComponent implémentent une méthode accept(). VISITOR n'impose pas à toutes les classes de la hiérarchie de posséder leur propre implémentation de cette méthode. Néanmoins, comme nous le verrons, toutes celles qui l'implémentent doivent apparaître sous la forme d'un argument dans une méthode visit() déclarée dans l'interface Machine-Visitor.

La méthode accept() de la classe MachineComponent est abstraite. Les deux sous-classes implémentent cette méthode en utilisant exactement le même code :

```
public void accept(MachineVisitor v) {  
    v.visit(this);  
}
```

Cette méthode étant identique dans les classes Machine et MachineComposite, vous pourriez vouloir la remonter dans la classe abstraite MachineComponent. Sachez toutefois que le compilateur voit une différence.

Exercice 29.1

Quelle différence un compilateur Java discerne-t-il entre les méthodes accept() des classes Machine et MachineComposite ? Ne regardez pas la solution avant d'avoir bien réfléchi car cette différence est essentielle pour comprendre VISITOR.

- *Les solutions des exercices de ce chapitre sont données dans l'Annexe B.*

L'interface MachineVisitor impose aux implementeurs de définir des méthodes pour "visiter" les machines et les composites de machines :

```
package com.oozinoz.machine;  
public interface MachineVisitor {  
    void visit(Machine m);  
    void visit(MachineComposite mc);  
}
```

Les méthodes accept() de MachineComponent combinées à l'interface Machine-Visitor invitent les développeurs à ajouter de nouvelles opérations à la hiérarchie.

Un VISITOR ordinaire

Imaginez que vous participiez au développement de la toute dernière unité de production d'Oozinoz à Dublin en Irlande. Les développeurs qui sont sur place ont créé un modèle objet pour la composition des machines et l'ont rendu accessible sous la forme de la méthode `dublin()` statique de la classe `OozinozFactory`. Pour afficher ce composite, ils ont créé une classe `MachineTreeModel` afin d'adapter les informations du modèle aux exigences d'un objet `JTree` (le code de `MachineTreeModel` se trouve dans le package `com.oozinoz.dublin`).

L'affichage des machines de l'unité de production demande de créer une instance de `MachineTreeModel` à partir du composite de l'unité et d'envelopper ce modèle dans des composants Swing :

```
package app.visitor;
import javax.swing.JScrollPane;
import javax.swing.JTree;
import com.oozinoz.machine.OozinozFactory;
import com.oozinoz.ui.SwingFacade;

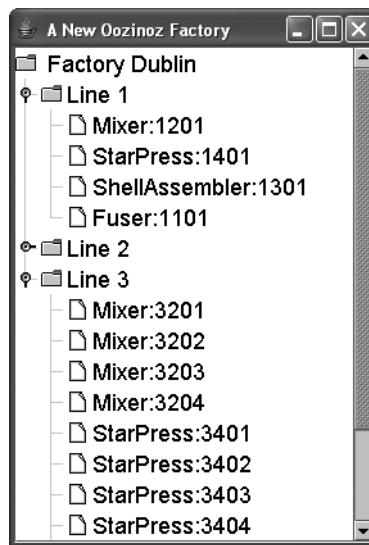
public class ShowMachineTreeModel {
    public ShowMachineTreeModel() {
        MachineTreeModel model = new MachineTreeModel(
            OozinozFactory.dublin());
        JTree tree = new JTree(model);
        tree.setFont(SwingFacade.getStandardFont());
        SwingFacade.launch(
            new JScrollPane(tree),
            " Une nouvelle unité de production Oozinoz");
    }

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

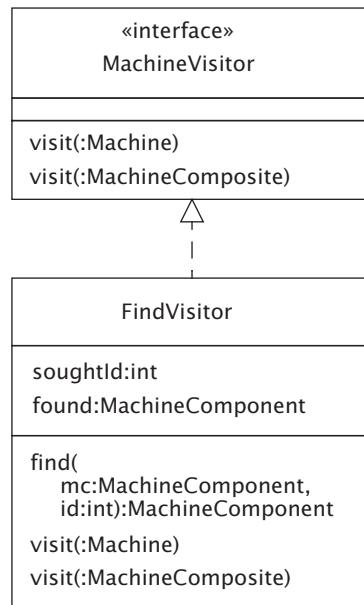
Ce programme affiche le résultat illustré Figure 29.2. Nombre de comportements utiles sont possibles pour un composite de machines. Par exemple, supposez que vous ayez besoin de trouver une certaine machine dans le modèle. Pour ajouter cette possibilité sans modifier la hiérarchie `MachineComponent`, vous pouvez créer une classe `FindVisitor`, comme le montre la Figure 29.3.

Figure 29.2

L'application GUI présente la composition des machines de la nouvelle unité de production irlandaise.

**Figure 29.3**

La classe *FindVisitor* ajoute une opération *find()* à la hiérarchie *Machine-Component*.



Les méthodes `visit()` ne retournent pas d'objet, aussi la classe `FindVisitor` enregistre-t-elle l'état d'une recherche dans sa variable d'instance `found` :

```
package app.visitor;

import com.oozinoz.machine.*;
import java.util.*;

public class FindVisitor implements MachineVisitor {
    private int soughtId;
    private MachineComponent found;

    public MachineComponent find(
        MachineComponent mc, int id) {
        found = null;
        soughtId = id;
        mc.accept(this);
        return found;
    }

    public void visit(Machine m) {
        if (found == null && m.getId() == soughtId)
            found = m;
    }

    public void visit(MachineComposite mc) {
        if (found == null && mc.getId() == soughtId) {
            found = mc;
            return;
        }
        Iterator iter = mc.getComponents().iterator();
        while (found == null && iter.hasNext())
            ((MachineComponent) iter.next()).accept(this);
    }
}
```

Les méthodes `visit()` examinent la variable `found` pour que la traversée de l'arbre se termine aussitôt que le composant recherché a été trouvé.

Exercice 29.2

Ecrivez un programme qui trouve et affiche l'objet `StarPress:3404` dans l'instance de `MachineComponent` renournée par `OozinozFactory.dublin()`.

La méthode `find()` ne se préoccupe pas de savoir si l'objet `MachineComponent` qu'elle reçoit est une instance de `Machine` ou de `MachineComposite`. Elle invoque simplement `accept()` qui invoque à son tour `visit()`.

Notez que la boucle dans la méthode `visit(:MachineComposite)` ne se soucie pas non plus de savoir si un composant enfant est une instance de `Machine` ou de `MachineComposite`. La méthode `visit()` invoque simplement l'opération `accept()` de chaque composant. La méthode qui s'exécute comme résultat de cette invocation dépend du type de l'enfant. La Figure 29.4 présente une séquence typique d'appels de méthodes.

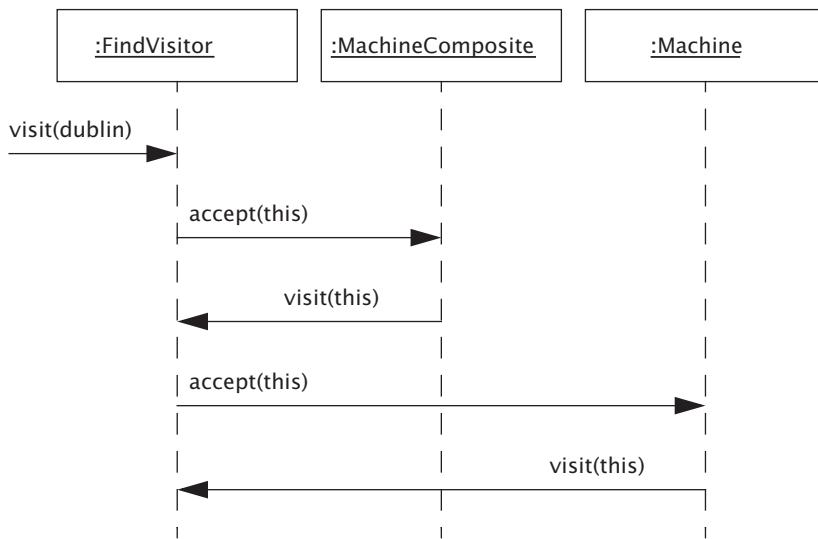


Figure 29.4

Un objet `FindVisitor` invoque une opération `accept()` pour déterminer quelle méthode `visit()` exécuter.

Lorsque la méthode `visit(:MachineComposite)` s'exécute, elle invoque donc l'opération `accept()` de chaque enfant du composite. Un enfant répond en invoquant l'opération `visit()` de l'objet `Visitor`. Comme le montre la Figure 29.4, cet aller-retour entre l'objet `Visitor` et l'objet qui reçoit l'invocation de `accept()` permet de récupérer le type de ce dernier. Cette technique, qualifiée de **double**

dispatching, garantit que la méthode `visit()` appropriée de la classe `Visitor` est exécutée.

Le double dispatching dans `VISITOR` permet de créer des classes visiteur avec des méthodes qui sont spécifiques aux divers types de la hiérarchie visitée. Vous pouvez ajouter pratiquement n'importe quel comportement au moyen de ce pattern, comme si vous contrôliez le code source. Comme autre exemple, considérez un visiteur qui trouve toutes les machines — les noeuds feuille — d'un composant-machine :

```
package app.visitor;
import com.oozinoz.machine.*;
import java.util.*;

public class RakeVisitor implements MachineVisitor {
    private Set leaves;

    public Set getLeaves(MachineComponent mc) {
        leaves = new HashSet();
        mc.accept(this);
        return leaves;
    }

    public void visit(Machine m) {
        // Exercice !
    }

    public void visit(MachineComposite mc) {
        // Exercice !
    }
}
```

Exercice 29.3

Complétez le code de la classe `RakeVisitor` pour collecter les feuilles (*leave*) d'un composant-machine.

Un court programme peut trouver les feuilles d'un composant-machine et les afficher :

```
package app.visitor;
```

```
import com.oozinoz.machine.*;
import java.io.*;
import com.oozinoz.filter.WrapFilter;

public class ShowRakeVisitor {
    public static void main(String[] args)
        throws IOException {
        MachineComponent f = OozinozFactory.dublin();
        Writer out = new PrintWriter(System.out);
        out = new WrapFilter(new BufferedWriter(out), 60);
        out.write(
            new RakeVisitor().getLeaves(f).toString());
        out.close();
    }
}
```

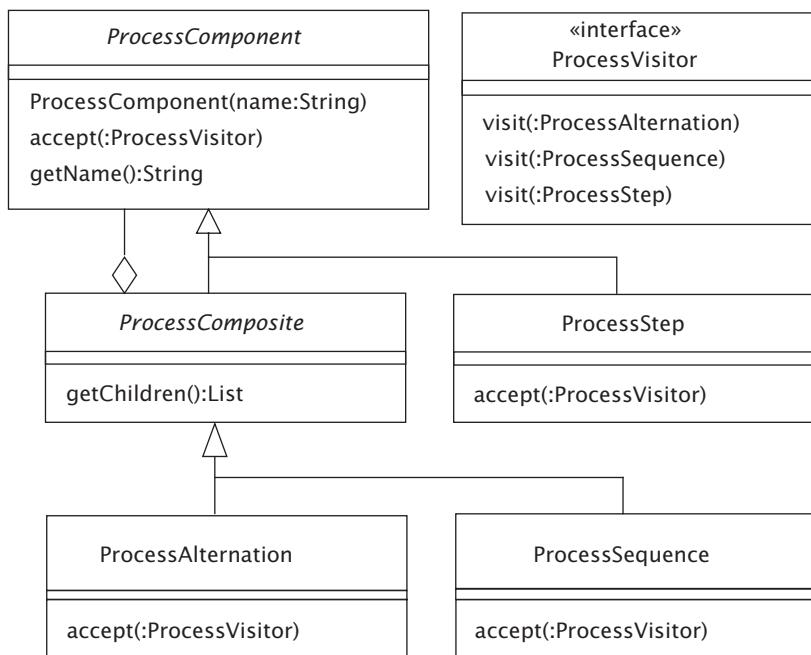
Ce programme utilise un filtre de passage à la ligne pour produire son résultat :

```
[StarPress:3401, Fuser:3102, StarPress:3402, Mixer:3202,
Fuser:3101, StarPress:3403, ShellAssembler:1301,
ShellAssembler:2301, Mixer:1201, StarPress:2401, Mixer:3204,
Mixer:3201, Fuser:1101, Fuser:2101, ShellAssembler:3301,
ShellAssembler:3302, StarPress:1401, Mixer:3203, Mixer:2202,
StarPress:3404, Mixer:2201, StarPress:2402]
```

Les classes `FindVisitor` et `RakeVisitor` ajoutent toutes deux un nouveau comportement à la hiérarchie `MachineComponent` et semblent fonctionner correctement. Cependant, le risque d'utiliser des visiteurs est qu'ils demandent de comprendre la hiérarchie qui est étendue. Un changement dans cette hiérarchie pourrait rendre votre visiteur inopérant, ou vous pourriez ne pas saisir correctement le fonctionnement de la hiérarchie. En particulier, vous pourriez avoir à gérer des cycles si le composite que vous visitez ne les empêche pas.

Cycles et VISITOR

La hiérarchie `ProcessComponent` qu'utilise Oozinoz pour modéliser ses flux de processus est une autre structure composite à laquelle il peut être utile d'intégrer un support de VISITOR. Contrairement aux composites de machines, il est naturel pour les flux de processus de contenir des cycles, et les objets visiteurs doivent éviter de générer des boucles infinies lorsqu'ils parcourront des composites de processus. La Figure 29.5 présente la hiérarchie `ProcessComponent`.

**Figure 29.5**

A l'instar de la hiérarchie `MachineComponent`, la hiérarchie `ProcessComponent` peut intégrer un support de `VISITOR`.

Supposez que vous vouliez afficher un composant-processus dans un format indenté. Dans le Chapitre 28, consacré au pattern `ITERATOR`, nous avons utilisé un itérateur pour afficher les étapes d'un flux de processus, que voici :

```

Fabriquer une bombe aérienne
Créer une coque interne
Contrôler
Retraiter la coque interne, ou finir la bombe
    Retraiter
    Désassembler
Terminer : ajouter la charge de propulsion, insérer le dispositif
d'allumage, et emballer

```

Retravailler une bombe implique de la désassembler et de la réassembler. L'étape qui suit Désassembler est Fabriquer une bombe aérienne. L'affichage précédent ne répète pas cette étape car l'itérateur voit qu'elle est déjà apparue une fois. Il serait néanmoins plus informatif de la faire apparaître de nouveau en indiquant que le

processus entre ici dans un cycle. Il serait également utile d'indiquer quels composites constituent des alternances par opposition à des séquences.

Pour créer un programme d'affichage des processus, vous pourriez créer une classe visiteur qui initialise un objet `StringBuilder` et ajoute à ce tampon les nœuds d'un composant-processus à mesure qu'ils sont visités. Pour indiquer qu'une étape d'un composite est une alternance, le visiteur pourrait faire précéder son nom d'un point d'interrogation (?). Pour indiquer qu'une étape est déjà apparue, il pourrait la faire suivre de trois points de suspension (...). Avec ces changements, le processus de fabrication d'une bombe aérienne ressemblerait à ce qui suit :

```
Fabriquer une bombe aérienne
    Créer une coque interne
    Contrôler
    ?Retraiter la coque interne, ou finir la bombe
        Retraiter
        Désassembler
        Fabriquer une bombe aérienne...
    Terminer : ajouter la charge de propulsion, insérer le dispositif
        d'allumage, et emballer
```

Un visiteur de composant-processus doit tenir compte des cycles, mais cela est facile à mettre en œuvre en utilisant un objet `Set` qui garde trace des nœuds visités. Le code de cette classe débuterait ainsi :

```
package app.visitor;
import java.util.List;
import java.util.HashSet;
import java.util.Set;
import com.oozinoz.process.*;

public class PrettyVisitor implements ProcessVisitor {
    public static final String INDENT_STRING = " ";
    private StringBuffer buf;
    private int depth;
    private Set visited;

    public StringBuffer getPretty(ProcessComponent pc) {
        buf = new StringBuffer();
        visited = new HashSet();
        depth = 0;
        pc.accept(this);
        return buf;
    }

    protected void printIndentedString(String s) {
        for (int i = 0; i < depth; i++)
            buf.append(INDENT_STRING);
```

```

        buf.append(s);
        buf.append("\n");
    }
    // ... méthodes visit()...
}

```

Cette classe utilise une méthode `getPretty()` pour initialiser les variables d'une instance et lancer l'algorithme visiteur. La méthode `printIndentedString()` gère l'indentation des étapes à mesure que l'algorithme explore plus avant un composite. Lorsqu'il visite un objet `ProcessStep`, le code affiche simplement le nom de l'étape :

```

public void visit(ProcessStep s) {
    printIndentedString(s.getName());
}

```

Vous avez peut-être remarqué dans la Figure 29.5 que la classe `ProcessComposite` n'implémente pas de méthode `accept()` mais que ses sous-classes le font. Visiter une alternance ou une séquence de processus requiert une logique identique, que voici :

```

public void visit(ProcessAlternation a) {
    visitComposite("?", a);
}

public void visit(ProcessSequence s) {
    visitComposite("", s);
}

protected void visitComposite(
    String prefix, ProcessComposite c) {
    if (visited.contains(c)) {
        printIndentedString(prefix + c.getName() + "...");
    } else {
        visited.add(c);
        printIndentedString(prefix + c.getName());
        depth++;

        List children = c.getChildren();
        for (int i = 0; i < children.size(); i++) {
            ProcessComponent child =
                (ProcessComponent) children.get(i);
            child.accept(this);
        }

        depth--;
    }
}

```

La différence entre visiter une alternance et visiter une séquence est que la première utilise un point d'interrogation comme préfixe. Dans les deux cas, si l'algorithme a déjà visité le nœud, nous affichons son nom suivi de trois points de suspension. Sinon, nous l'ajoutons à une collection de nœuds visités, affichons son préfixe — un point d'interrogation ou rien — et "acceptons" ses enfants. Chose courante avec le pattern VISITOR, le code recourt au polymorphisme pour déterminer si les nœuds enfant sont des instances des classes `ProcessStep`, `ProcessAlternation` ou `ProcessSequence`.

Un court programme peut maintenant afficher le flux de processus :

```
package app.visitor;

import com.oozinoz.process.ProcessComponent;
import com.oozinoz.process.ShellProcess;

public class ShowPrettyVisitor {
    public static void main(String[] args) {
        ProcessComponent p = ShellProcess.make();
        PrettyVisitor v = new PrettyVisitor();
        System.out.println(v.getPretty(p));
    }
}
```

Ce programme affiche le résultat suivant :

```
Fabriquer une bombe aérienne
Créer la coque interne
Contrôler
?Retraiter la coque interne, ou finir la bombe
    Retraiter
        Désassembler
            Fabriquer une bombe aérienne...
Terminer : ajouter la charge de propulsion, insérer le dispositif
d'allumage, et emballer
```

Ce résultat est plus informatif que celui obtenu en parcourant simplement le modèle de processus. Le point d'interrogation qui apparaît signale que les étapes de ce composite sont des alternances. De plus, le fait d'afficher l'étape `Fabriquer une bombe aérienne` une seconde fois, suivie de points de suspension, est plus clair que de simplement omettre une étape qui se répète.

Les développeurs de la hiérarchie `ProcessComponent` intègrent le support de VISITOR en y incluant des méthodes `accept()` et en définissant l'interface `ProcessVisitor`. Ils ont tous conscience de la nécessité d'éviter les boucles infinies lors de l'itération sur des flux de processus. Comme le montre la classe

PrettyVisitor, ils doivent aussi avoir conscience de l'éventualité de cycles dans les composants-processus. Ils pourraient réduire le risque d'erreur en prévoyant un support des cycles dans le cadre de leur support de VISITOR.

Exercice 29.4

Comment les développeurs de ProcessComponent peuvent-ils intégrer à la hiérarchie à la fois un support des cycles et un support de VISITOR ?

Risques de VISITOR

VISITOR est un pattern sujet à controverse. Certains développeurs évitent systématiquement de l'appliquer, tandis que d'autres défendent son utilisation et suggèrent des moyens de le renforcer, bien que ces suggestions augmentent généralement la complexité du code. Le fait est que ce pattern peut donner lieu à de nombreux problèmes de conception.

La fragilité de VISITOR est perceptible dans les exemples de ce chapitre. Par exemple, les développeurs de la hiérarchie MachineComponent ont choisi de faire une distinction entre les nœuds Machine et les nœuds MachineComposite, mais ils ne différencient pas les sous-classes de Machine. Si vous aviez besoin de distinguer différents types de machines dans votre visiteur, il vous faudrait procéder à une vérification du type ou employer d'autres techniques pour déterminer quel type de machine une méthode `visit()` a reçue. Vous pensez peut-être que les développeurs auraient dû inclure tous les types de machines ainsi qu'une méthode `visit(:Machine)` générique dans l'interface visiteur. Mais de nouveaux types de machines apparaissent régulièrement et cette solution ne serait donc pas plus robuste.

VISITOR ne représente pas toujours une bonne option selon les changements susceptibles d'intervenir. Si la hiérarchie est stable et que les comportements associés changent, ce choix peut convenir. Mais si les comportements sont stables et que la hiérarchie change, cela vous obligera à actualiser les visiteurs existants pour qu'ils puissent supporter les nouveaux types de nœuds.

Un autre exemple de faiblesse apparaît dans la hiérarchie ProcessComponent. Les développeurs savent que les cycles représentent un danger lié aux modèles de flux

de processus. Comment peuvent-ils communiquer ce risque aux développeurs de visiteurs ?

Ces difficultés sont révélatrices du problème fondamental de ce pattern, à savoir que l'extension des comportements d'une hiérarchie demande normalement une connaissance experte de sa conception. Si vous ne possédez pas cette expertise, vous risquez de tomber dans un piège, comme celui de ne pas éviter les cycles d'un flux de processus. Vous risquez de créer des dépendances dangereuses qui ne résisteront pas aux changements de la hiérarchie. La distribution de l'expertise et du contrôle du code source peut rendre VISITOR dangereux à appliquer.

Un cas classique où ce pattern semble bien fonctionner sans causer de problèmes subséquents est celui des analyseurs syntaxiques des langages informatiques. Leurs développeurs s'arrangent souvent pour que l'analyseur (*parser*) crée un **arbre syntaxique abstrait**, c'est-à-dire une structure qui organise le texte en entrée en fonction de la grammaire du langage. Vous pourriez développer une variété de comportements pour accompagner ces arbres, et le pattern VISITOR représente une approche efficace permettant cela. Dans ce cas classique, il n'y a quasiment pas de comportements, voire aucun, dans la hiérarchie visitée. La responsabilité de concevoir les comportements revient aux visiteurs, évitant ainsi la distribution de responsabilité observée dans les exemples de ce chapitre.

Comme n'importe quel autre pattern, VISITOR n'est jamais nécessaire, sinon il apparaît partout où il le serait. En outre, il existe souvent des alternatives offrant des conceptions plus robustes.

Sachez qu'il est le plus performant dans les conditions suivantes :

- L'ensemble des types de nœuds est stable.
- Un changement courant est l'ajout de nouvelles fonctions qui s'appliquent aux différents nœuds.

A noter que les nouvelles fonctions devraient toucher tous les types de nœuds.

Exercice 29.5

Enumérez deux alternatives à l'emploi de VISITOR dans les hiérarchies de machines et de processus d'Oozinoz.

Résumé

Le pattern VISITOR permet de définir une nouvelle opération pour une hiérarchie sans avoir à modifier ses classes. Son application implique de définir une interface pour les visiteurs et d'ajouter des méthodes `accept()` à la hiérarchie qui sera appellée par un visiteur. Ces méthodes renvoient leur appel au visiteur en utilisant une technique de double dispatching, laquelle fait qu'une méthode `visit()` s'appliquera au type d'objet correspondant dans la hiérarchie.

Le développeur d'un visiteur doit connaître certaines, voire toutes les subtilités de conception de la hiérarchie visitée. En particulier, les visiteurs doivent avoir connaissance des cycles susceptibles de survenir dans le modèle objet visité. Ce genre de difficultés pousse certains développeurs à renoncer à utiliser VISITOR, et à lui préférer d'autres alternatives. La décision d'employer ce pattern devrait dépendre de votre philosophie de conception, de votre équipe et des spécificité de votre application.

VI

Annexes

A

Recommandations

Si vous avez lu ce livre jusqu'ici, félicitations ! Si vous avez effectué tous les exercices, bravo ! Vous avez développé une bonne connaissance pratique des patterns de conception. Cette annexe vous donne des recommandations pour pousser plus loin votre apprentissage.

Tirer le meilleur parti du livre

Si vous n'avez *pas* accompli les exercices du livre, vous n'êtes pas le seul ! Nous sommes tous très occupés, et il est tentant de simplement réfléchir au problème puis de consulter la solution. Cette démarche est assez commune, mais ce qui est dommage, c'est que vous avez la possibilité de devenir un développeur hors norme. Refaites les exercices un à un et sérieusement, en vous reportant à la solution uniquement lorsque vous pensez avoir trouvé une bonne réponse ou si vous êtes bloqué. Faites-le *maintenant*. Ne vous dites pas que vous aurez davantage de temps plus tard. En exerçant vos connaissances fraîchement acquises sur les patterns, vous acquerrez l'assurance dont vous avez besoin pour commencer à les appliquer dans votre travail.

Nous vous suggérons également de télécharger le code disponible à l'adresse www.oozinoz.com et de reproduire les exemples du livre sur votre système. Parvenir à exécuter ce code vous apportera plus de confiance que si vous vous contentez de travailler sur papier. Vous pouvez aussi élaborer de nouveaux exercices. Vous voudrez peut-être trouver de nouvelles façons de combiner des filtres DECORATOR, ou implémenter un ADAPTER de données qui affiche des informations d'un domaine familier.

A mesure que vous vous familiarisez avec les patterns, vous devriez commencer à comprendre les exemples classiques de leur utilisation. Vous commencerez également à identifier les endroits où il peut être approprié d'appliquer des patterns dans votre code.

Connaître ses classiques

Les patterns rendent souvent une conception plus robuste. Cette idée n'est pas nouvelle, aussi n'est-il pas surprenant que de nombreux patterns soient intégrés aux bibliothèques de classes Java. Si vous pouvez repérer un pattern dans le corps d'un programme, vous pouvez saisir la conception et la communiquer à d'autres développeurs qui comprennent également les patterns. Par exemple, si un développeur comprend comment fonctionne DECORATOR, cela a du sens d'expliquer que les flux Java sont des décorateurs.

Voici un test pour contrôler votre compréhension des exemples classiques d'utilisation des patterns, qui apparaissent dans Java et ses bibliothèques .

- Comment les GUI contrôlent-elles l'emploi du pattern OBSERVER ?
- Pourquoi les menus emploient-ils souvent le pattern COMMAND ?
- En quoi les drivers constituent-ils un bon exemple du pattern BRIDGE ? Chaque driver est-il une instance du pattern ADAPTER ?
- Que signifie le fait de dire que les flux Java utilisent le pattern DECORATOR ?
- Pourquoi le pattern PROXY est-il essentiel pour la conception de RMI ?
- Si le tri est un bon exemple du pattern TEMPLATE METHOD, quelle étape de l'algorithme n'est pas spécifiée initialement ?

L'idéal est de répondre à ces questions sans vous aider du livre. Un bon exercice est de mettre vos réponses par écrit et de les partager avec des collègues.

Appliquer les patterns

Devenir un meilleur développeur est ce qui incite le plus souvent à apprendre les patterns de conception. Leur application peut se faire de deux manières : lorsque vous ajoutez du code ou *via* une refactorisation. Si une partie de votre code est complexe et difficile à gérer, vous pourriez l'améliorer en le refactorisant et en

utilisant un pattern. Avant de vous lancer dans un tel projet, assurez-vous que cela en vaille la peine. Veillez aussi à créer une suite de tests automatisés pour le code que vous réorganisez.

Supposez maintenant que vous comprenez bien les patterns étudiés et soyez déterminé à les utiliser prudemment et de manière appropriée. Comment trouvez-vous des opportunités ? En fait, des occasions se présentent assez fréquemment. Pour les identifier, considérez les points suivants .

- Votre base de code comporte-t-elle des parties complexes liées à l'état d'un système ou de l'utilisateur de l'application ? Si oui, vous pourriez l'améliorer en appliquant le pattern STATE.
- Votre code combine-t-il la sélection d'une stratégie et l'exécution de cette stratégie ? Si oui, vous pourriez l'améliorer en appliquant le pattern STRATEGY.
- Votre client ou analyste vous remet-il des organigrammes qui donnent lieu à du code difficile à comprendre ? Si oui, vous pouvez appliquer le pattern INTERPRETER, en faisant de chaque noeud de l'organigramme une instance d'une classe de la hiérarchie interpréteur. Vous obtiendrez ainsi une traduction directe de l'organigramme en code.
- Votre code comporte-t-il un composite faible qui n'autorise pas ses enfants à être eux-mêmes des composites ? Vous pourriez renforcer ce code à l'aide du pattern COMPOSITE.
- Avez-vous rencontré des erreurs d'intégrité relationnelle dans votre modèle objet ? Vous pourriez les éviter en appliquant le pattern MEDIATOR pour centraliser la modélisation des relations entre les objets.
- Votre code comporte-t-il des endroits où des clients utilisent les informations d'un service pour décider quelle classe instancier ? Vous pourriez améliorer et simplifier ce code en appliquant le pattern FACTORY METHOD.

Connaître les patterns permet de développer un vocabulaire riche d'idées de conception. Si vous recherchez des opportunités, vous n'aurez probablement pas à attendre longtemps avant de trouver une conception qui peut être améliorée par l'application d'un pattern. Mais ne forcez pas les choses.

Continuer d'apprendre

C'est déjà une bonne chose que vous ayez eu l'opportunité, la volonté et l'ambition d'acquérir et de lire ce livre. Continuez ainsi ! Déterminez combien d'heures par semaine vous voulez consacrer à votre carrière. Si vous arrivez à vous en ménager cinq, c'est très bien. Passez ce temps en dehors de votre bureau, à lire des livres et des magazines ou à écrire des programmes en rapport avec un sujet qui vous intéresse. Que cela devienne une habitude aussi régulière que d'aller au bureau. Traitez sérieusement cet aspect de votre carrière et vous deviendrez un bien meilleur développeur. Vous n'en apprécieriez ensuite que mieux votre travail.

Vous pouvez prendre de nombreuses directions, l'essentiel étant de continuer à progresser. Considérez l'acte d'apprendre comme faisant partie de votre plan de carrière, et adonnez-vous aux sujets qui vous passionnent le plus. Pensez aux compétences que vous pouvez acquérir avec le temps, et entreprenez ce qu'il faut pour cela.

Steve.Metsker@acm.org

William.Wake@acm.org

B

Solutions

Introduction aux interfaces

Solution 2.1

Une classe abstraite ne contenant aucune méthode non abstraite est semblable à une interface du point de vue de son utilité. Notez toutefois les nuances suivantes :

- Une classe peut implémenter autant d'interfaces que nécessaire mais elle ne peut étendre au plus qu'une classe abstraite.
- Une classe abstraite peut avoir des méthodes non abstraites, mais toutes les méthodes d'une interface sont abstraites.
- Une classe abstraite peut déclarer et utiliser des champs alors qu'une interface ne le peut pas, bien qu'elle puisse créer des constantes `static final`.
- Une classe abstraite peut avoir des méthodes déclarées `public`, `protected`, `private` ou sans accès (package). Les méthodes d'une interface ont un accès implicitement public.
- Une classe abstraite peut définir des constructeurs alors qu'une interface ne le peut pas.

Solution 2.2

- A. *Vrai.* Les méthodes d'une interface sont toujours abstraites même sans déclaration explicite.
- B. *Vrai.* Les méthodes d'une interface sont publiques même sans déclaration explicite.
- C. *Faux.* La visibilité d'une interface peut se limiter au package dans lequel elle réside. Dans ce cas, elle est marquée `public` afin que les classes externes à `com.oozinoz.simulation` puissent y accéder.
- D. *Vrai.* Par exemple, les interfaces `List` et `Set` étendent toutes deux l'interface `Collection` dans `java.util`.
- E. *Faux.* Une interface sans méthode est appelée *interface de marquage*. Parfois, une méthode haut dans la hiérarchie de classe, telle que `Object.clone()`, n'est pas appropriée pour toutes les sous-classes. Vous pouvez créer une interface de marquage qui demande aux sous-classes d'opter ou non pour une telle stratégie. La méthode `clone()` sur `Object` requiert que les sous-classes optent pour la stratégie, en déclarant qu'elles implémentent l'interface de marquage `Cloneable`.
- F. *Faux.* Une interface ne peut déclarer des champs d'instance, bien qu'elle puisse créer des constantes en déclarant des champs qui sont `static` et `final`.
- G. *Faux.* Ce peut être une bonne idée mais une interface ne dispose d aucun moyen pour exiger que les classes l'implémentant fournissent un certain constructeur.

Solution 2.3

Un tel exemple se produit lorsque des classes peuvent être enregistrées en tant que listeners d'événements. Les classes reçoivent des notifications pour leur propre compte et non de celui de l'appelant. Par exemple, nous pourrions réagir avec `MouseListener.mouseDragged()` mais avoir un corps vide pour `Listener.mousePositionMoved()` pour le même listener.

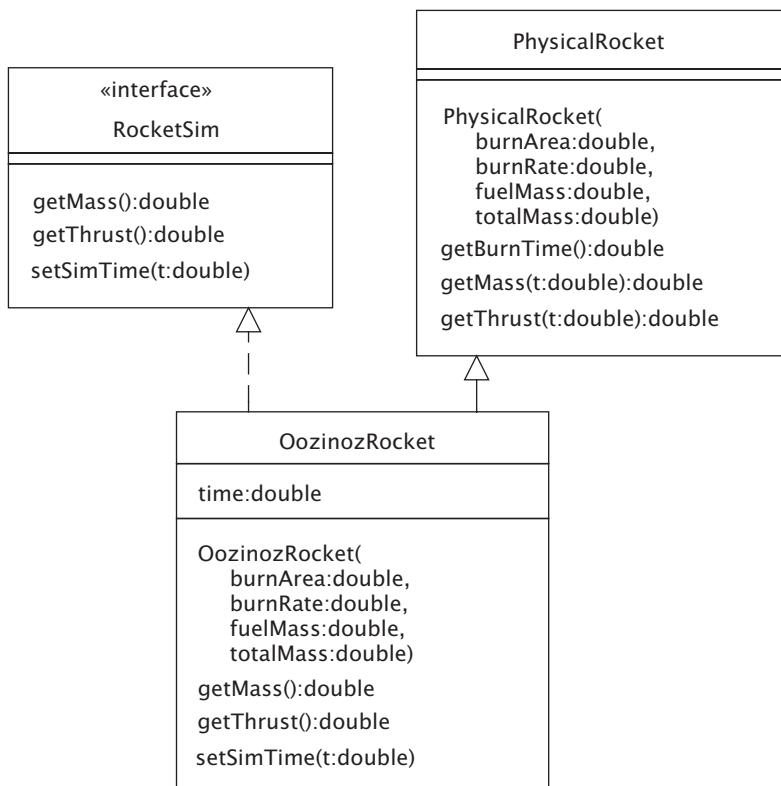
ADAPTER

Solution 3.1

Votre solution devrait ressembler au diagramme illustré à la Figure B.1.

Figure B.1

La classe OozinozRocket adapte la classe PhysicalRocket pour répondre aux besoins déclarés dans l'interface RocketSim.



Les instances de la classe OozinozRocket peuvent fonctionner en tant qu'objets PhysicalRocket ou RocketSim. Le pattern ADAPTER vous permet d'adapter les méthodes que vous avez à celles dont un client a besoin.

Solution 3.2

Le code pour compléter la classe devrait être :

```

package com.oozinoz.firework;
import com.oozinoz.simulation.*;
public class OozinozRocket
    extends PhysicalRocket implements RocketSim {
    private double time;
    public OozinozRocket(
        double burnArea,
        double burnRate,
        double fuelMass,
        double totalMass) {
        super(burnArea, burnRate, fuelMass, totalMass);
    }
    ...
}
  
```

```

public double getMass() {
    return getMass(time);
}
public double getThrust() {
    return getThrust(time);
}
public void setSimTime(double time) {
    this.time = time;
}
}

```

Vous trouverez ce code dans le package `com.oozinoz.firework` du code source compagnon de ce livre.

Solution 3.3

La Figure B.2 illustre une solution.

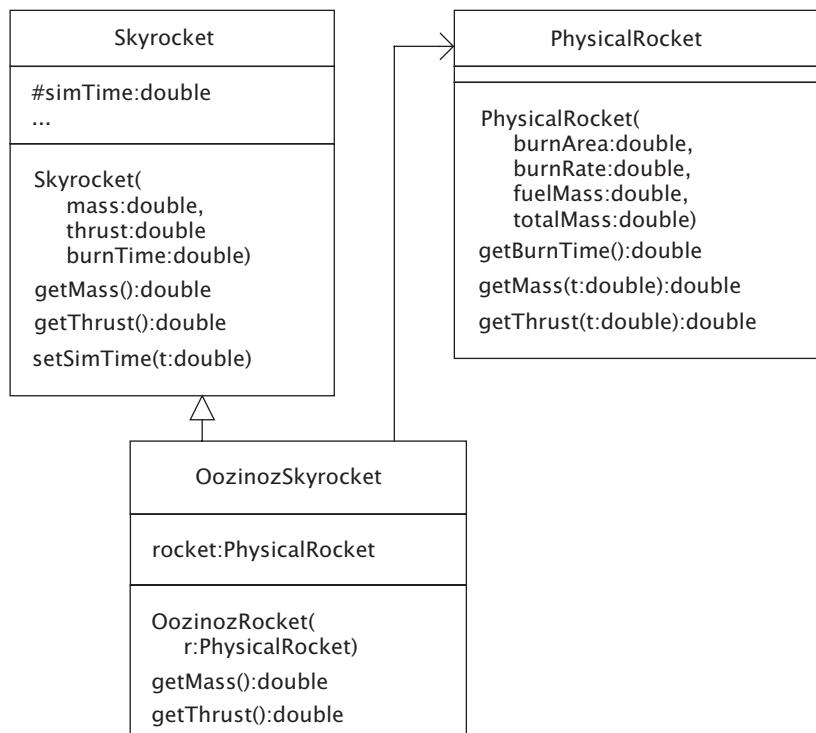


Figure B.2

Un objet `OozinozSkyrocket` est un objet `Skyrocket`, mais son travail est réalisé par transmission des appels à un objet `PhysicalRocket`.

La classe `OozinozSkyrocket` est un adaptateur d'objet. Elle étend `Skyrocket` pour qu'un objet `OozinozSkyrocket` puisse fonctionner où un objet `Skyrocket` est requis.

Solution 3.4

La conception avec adaptateur d'objet utilisée par la classe `OozinozSkyrocket` peut se révéler plus fragile qu'une approche par adaptateur de classe pour les raisons suivantes :

- Il n'y a aucune spécification de l'interface que la classe `OozinozSkyrocket` fournit. Par conséquent, même si cela n'a pas été détecté à la compilation, `Skyrocket` pourrait changer d'une certaine façon qui pourrait être source de problèmes lors de l'exécution.
- La classe `OozinozSkyrocket` compte sur la capacité à accéder à la variable `simTime` de sa classe parent, bien qu'il n'y ait aucune garantie que cette variable sera toujours déclarée comme étant protégée et aucune certitude quant à la signification de ce champ dans la classe `Skyrocket`. Nous n'attendons pas de la part des fournisseurs du code qu'ils s'écartent de leur conception pour changer le code `Skyrocket` sur lequel nous nous appuyons, mais nous disposons toutefois d'un contrôle limité sur ce qu'ils font.

Solution 3.5

Votre code pourrait ressembler à l'exemple suivant :

```
package app.adapter;
import javax.swing.table.*;
import com.oozinoz.firework.Rocket;
public class RocketTable extends AbstractTableModel {
    protected Rocket[] rockets;
    protected String[] columnNames = new String[] {
        "Name", "Price", "Apogee" };
    public RocketTable(Rocket[] rockets) {
        this.rocks = rockets;
    }
    public int getColumnCount() {
        return columnNames.length;
    }
    public String getColumnName(int i) {
        return columnNames[i];
    }
    public int getRowCount() {
        return rockets.length;
    }
}
```

```

public Object getValueAt(int row, int col) {
    switch (col) {
        case 0:
            return rockets[row].getName();
        case 1:
            return rockets[row].getPrice();
        case 2:
            return new Double(rockets[row].getApogee());
        default:
            return null;
    }
}
}

```

L’interface `TableModel` est un bon exemple de l’efficacité de prévoir à l’avance une adaptation. Cette interface, ainsi que son implémentation partielle `AbstractTableModel`, limite le travail nécessaire pour montrer les objets spécifiques dans une table de GUI. Votre solution devrait permettre de constater comme il est simple de recourir à l’adaptation lorsqu’elle est supportée par une interface.

Solution 3.6

- *Un argument pour.* Lorsque l’utilisateur clique avec la souris, je dois traduire, ou adapter, l’appel Swing résultant en une action appropriée. En d’autres termes, lorsque je dois adapter des événements de GUI à l’interface de mon application, j’emploie les classes d’adaptation de Swing. Je réalise une conversion d’une interface à une autre, c’est-à-dire l’objectif du pattern ADAPTER.
- *Un argument contre.* Les classes d’adaptation dans Swing sont des stubs : elles n’adaptent rien. Vous dérivez des sous-classes de ces classes, où vous implémentez alors les méthodes qui doivent réaliser quelque chose. Ce sont vos méthodes et votre classe qui appliqueront éventuellement le pattern ADAPTER. Si “l’adaptateur” de Swing avait été nommé disons `DefaultMouseListener`, cet argument ne pourrait être avancé.

FACADE

Solution 4.1

Voici quelques différences notables entre une démo et une façade :

- Une démo est généralement une application autonome alors qu’une façade ne l’est généralement pas.

- Une démo inclut généralement un échantillon, ce qu'une façade ne fait pas.
- Une façade est généralement configurable, pas une démo.
- Une façade est prévue pour être réutilisée, pas une démo.
- Une façade est prévue pour être utilisée en production, pas une démo.

Solution 4.2

La classe `JOptionPane` est l'un des exemples peu nombreux d'une façade dans les bibliothèques de classes Java. Elle est utilisable en production, configurable et est prévue pour être réutilisée. Avant toute chose, `JOptionPane` réalise l'objectif du pattern **FAÇADE** en fournissant une interface simple qui facilite l'emploi d'une classe `JDialog`. Vous pouvez soutenir qu'une façade simplifie un "sous-système" et que la classe `JDialog` seule ne remplit pas les conditions pour être un sous-système. C'est toutefois la richesse des fonctionnalités de cette classe qui donne toute sa valeur à une façade.

Sun Microsystems incorpore de nombreuses démos dans son JDK. Ces classes ne font toutefois jamais partie des bibliothèques de classes Java. C'est-à-dire qu'elles n'apparaissent pas dans les packages nommés avec le préfixe `java`. Une façade peut appartenir aux bibliothèques Java, mais pas les démos.

La classe `JOptionPane` possède des dizaines de méthodes statiques qui font effectivement d'elle un utilitaire ainsi qu'une application de **FAÇADE**. Toutefois, à strictement parler, elle ne remplit pas les conditions de la définition dans UML d'un utilitaire, laquelle exige que toutes les méthodes soient statiques.

Solution 4.3

Voici quelques points de vue acceptables mais opposés concernant la pénurie de façades dans les bibliothèques de classes Java.

- En tant que développeur Java, vous êtes bien avisé de développer une profonde connaissance des outils de la bibliothèque. Les façades limitent nécessairement la façon dont vous pouvez appliquer n'importe quel système. Elles seraient une distraction et un élément éventuel de méprise dans les bibliothèques dans lesquelles elles apparaîtraient.
- Les caractéristiques d'une façade se situent quelque part entre la richesse d'un kit d'outils et la spécificité d'une application. Créer une façade requiert d'avoir

une certaine idée du type d'applications qu'elle supportera. Il est impossible de prévoir cela compte tenu de l'immense diversité du public utilisateur des bibliothèques Java.

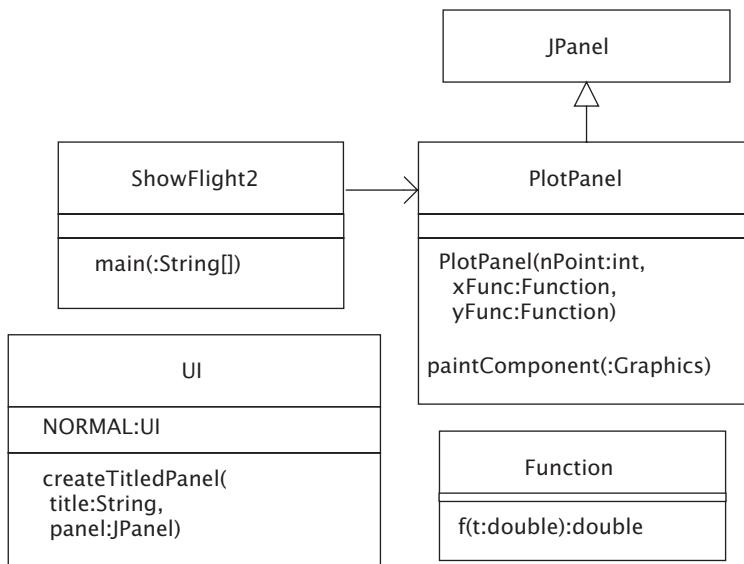
- La présence rare de façades dans les bibliothèques Java est un point faible. L'ajout de davantage de façades serait fortement utile.

Solution 4.4

Votre réponse devrait s'apparenter au diagramme de la Figure B.3.

Figure B.3

Ce diagramme illustre l'application de calcul de trajectoire restructurée en classes se chargeant chacune d'une tâche.



Notez que `createTitledPanel()` n'est pas une méthode statique. Votre solution a-t-elle prévu que ces méthodes soient statiques ? Pourquoi ou pourquoi pas ?

Le code présenté dans ce livre choisit les méthodes de UI comme étant non statiques pour qu'une sous-classe de UI puisse les redéfinir, créant ainsi un kit différent pour élaborer des interfaces graphiques utilisateur. Pour permettre la disponibilité d'une interface utilisateur standard, cette conception se réfère à l'objet singleton NORMAL.

Voici le code approprié pour la classe UI :

```

public class UI {
    public static final UI NORMAL = new UI();
}

```

```
protected Font font =
    new Font("Book Antiqua", Font.PLAIN, 18);
// beaucoup de choses omises
public Font getFont() {
    return font;
}
public TitledBorder createTitledBorder(String title) {
    TitledBorder border =
        BorderFactory.createTitledBorder(
            BorderFactory.createBevelBorder(
                BevelBorder.RAISED),
            title,
            TitledBorder.LEFT,
            TitledBorder.TOP);
    border.setTitleColor(Color.black);
    border.setTitleFont(getFont());
    return border;
}
public JPanel createTitledPanel(
    String title, JPanel in) {
    JPanel out = new JPanel();
    out.add(in);
    out.setBorder(createTitledBorder(title));
    return out;
}
```

Voyez le Chapitre 17 pour plus d'informations sur la conception de kits de GUI, et le Chapitre 8 qui étudie les singltons.

COMPOSITE

Solution 5.1

Prévoir la classe Composite de façon à pouvoir conserver une collection d'objets Component demande qu'un objet Composite puisse contenir des objets feuilles ou d'autres objets composites.

En d'autres termes, cette conception nous permet de modéliser des groupes en tant que collections d'autres groupes. Par exemple, nous pouvons définir les priviléges système d'un utilisateur sous forme d'une collection de priviléges spécifiques ou d'autres groupes de priviléges. Nous pourrions aussi définir une procédure en tant que collection d'étapes procédurales et d'autres procédures. De telles définitions sont beaucoup plus souples que la définition d'un composite comme devant être une collection de feuilles uniquement.

Si vous n'autorisez que des collections de feuilles, les composites ne pourront avoir qu'un niveau de profondeur.

Solution 5.2

Pour la classe Machine, getMachineCount() devrait ressembler à l'exemple suivant :

```
public int getMachineCount() {
    return 1;
}
```

Le diagramme de classes montre que MachineComposite utilise un objet List pour garder trace de ses composants. Pour compter les machines dans un composite, vous pourriez écrire le code suivant :

```
public int getMachineCount() {
    int count = 0;
    Iterator i = components.iterator();
    while (i.hasNext()) {
        MachineComponent mc = (MachineComponent) i.next();
        count += mc.getMachineCount();
    }
    return count;
}
```

Si vous utilisez le JDK 1.5, vous pourriez utiliser une boucle for étendue.

Solution 5.3

<i>Méthode</i>	<i>Classe</i>	<i>Définition</i>
getMachineCount()	MachineComposite	Retourne la somme des comptes pour chaque composant de Component
	Machine	Retourne 1
isCompletelyUp()	MachineComposite	Retourne true si tous les composants sont actifs
	Machine	Retourne true si cette machine est active
stopAll()	MachineComposite	Indique à tous les composants de tout arrêter
	Machine	Arrête cette machine

<i>Méthode</i>	<i>Classe</i>	<i>Définition</i>
getOwners()	MachineComposite	Crée un set (ensemble), pas une liste ; ajoute les propriétaires de tous les composants, puis retourne le set
	Machine	Retourne les propriétaires de cette machine
getMaterial()	MachineComposite	Retourne une collection de tous les produits sur les composants-machine
	Machine	Retourne les produits se trouvant sur cette machine

Solution 5.4

Le programme affiche :

```
Nombre de machines : 4
```

Il n'y a en fait que trois machines dans `plant`, mais la machine `mixer` est comptée par `plant` et `bay`. Ces deux objets contiennent une liste de composants-machine qui réfèrent à `mixer`.

Les résultats pourraient être pires. Supposez qu'un ingénieur ajoute l'objet `plant` en tant que composant du composite `bay`, un appel de `getMachineCount()` entrerait dans une boucle infinie.

Solution 5.5

Voici une implémentation raisonnable de `MachineComposite.isTree()` :

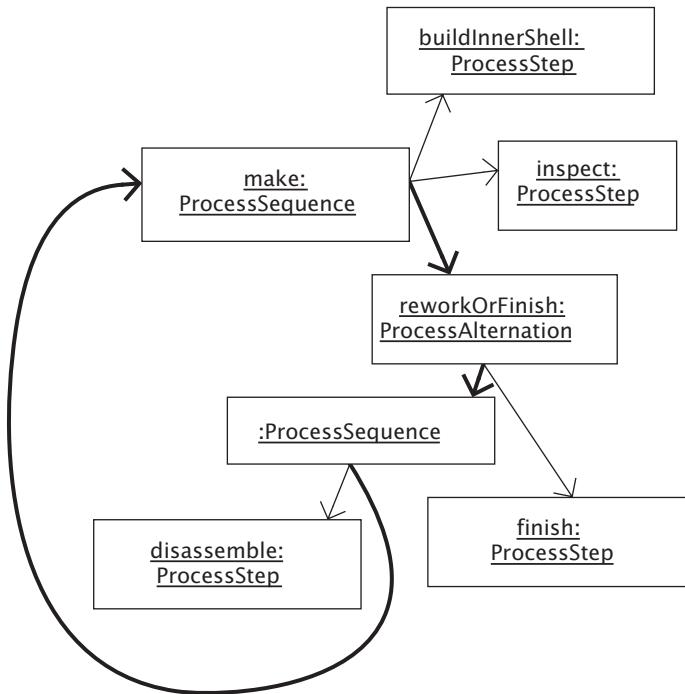
```
protected boolean isTree(Set visited) {
    visited.add(this);
    Iterator i = components.iterator();
    while (i.hasNext()) {
        MachineComponent c = (MachineComponent) i.next();
        if (visited.contains(c) || !c.isTree(visited))
            return false;
    }
    return true;
}
```

Solution 5.6

Votre solution devrait montrer les liens de la Figure B.4.

Figure B.4

Les lignes épaisses dans ce diagramme objet signalent le cycle inhérent au processus de fabrication.



BRIDGE

Solution 6.1

Pour contrôler diverses machines avec une interface commune, vous pouvez appliquer le pattern ADAPTER créant une classe d'adaptation pour chaque contrôleur. Chaque classe peut traduire les appels d'interface standard en appels gérés par les contrôleurs existants.

Solution 6.2

Votre code pourrait être comme suit :

```

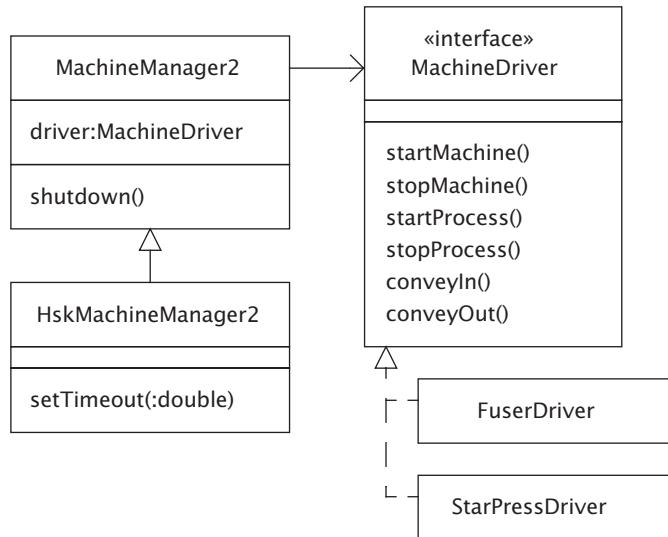
public void shutdown() {
    stopProcess();
    conveyOut();
    stopMachine();
}
  
```

Solution 6.3

La Figure B.5 illustre une solution.

Figure B.5

Ce diagramme représente une abstraction — une hiérarchie de types de gestionnaires de machine — distincte des implémentations de l'objet abstrait *driver* utilisé par l'abstraction.



Solution 6.4

La Figure B.6 suggère une solution.

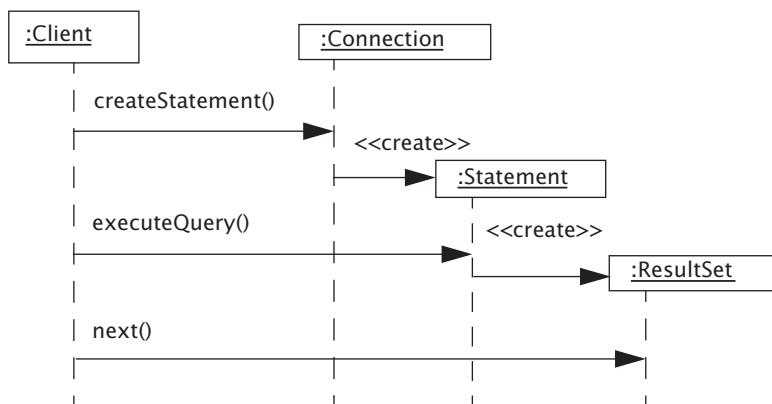


Figure B.6

Ce diagramme illustre le flux de messages principal intervenant dans une application JDBC.

Solution 6.5

Voici deux arguments en faveur de l'écriture de code spécifique à SQL Server :

1. Nous ne pouvons prévoir le futur. Aussi, investir de l'argent maintenant en prévision d'éventualités qui pourraient ne jamais se produire est une erreur classique. Nous disposons de SQL Server maintenant, et davantage de vitesse signifie de meilleurs temps de réponse, ce qui signifie des fonds en banque aujourd'hui.
2. En nous engageant exclusivement pour SQL Server, nous pouvons utiliser toutes les fonctionnalités offertes par la base de données, sans avoir à nous inquiéter de savoir si d'autres drivers de base de données les supporteront.

Voici deux arguments en faveur de l'emploi de drivers SQL génériques :

1. Si nous utilisons des drivers SQL génériques pour écrire du code, il sera plus facile de le modifier si nous changeons de fournisseur de base de données et commençons à utiliser, par exemple, Oracle. En verrouillant le code pour SQL Server, nous limitons notre capacité à tirer parti de l'offre variée du marché des bases de données.
2. L'emploi de drivers génériques nous permet d'écrire du code expérimental pouvant s'exécuter avec des bases de données peu onéreuses, telles que MySql, sans s'appuyer sur un test spécifique dans SQL Server.

Introduction à la responsabilité

Solution 7.1

Voici quelques problèmes que posent le diagramme :

- La méthode `Rocket.thrust()` retourne un objet `Rocket` à la place d'un type quelconque de nombre ou de quantité physique.
- La classe `LiquidRocket` possède une méthode `getLocation()` bien que rien dans le diagramme ou le domaine de problèmes ne suggère que les fusées doivent avoir un emplacement. Même si nous l'avions fait, il n'y a aucune raison pour que les fusées à combustible liquide — pas les autres objets `Rocket` — aient un emplacement.
- La méthode `isLiquid()` peut constituer une alternative acceptable à l'emploi de l'opérateur `instanceof`, mais nous nous attendrions alors aussi à ce que la super-classe ait une méthode `isLiquid()` retournant `false`.

- CheapRockets (*fusées bon marché*) est un nom pluriel bien que les noms de classes soient par convention au singulier.
- La classe CheapRockets implémente Runnable, bien que cette interface n'ait pas affaire avec les objets CheapRockets du domaine de problèmes.
- Nous pourrions modéliser l'aspect bon marché avec seulement des attributs. Il n'y a donc aucune justification à la création d'une classe simplement pour des fusées bon marché.
- La classe CheapRockets introduit une structuration du code qui entre en conflit avec la structuration du modèle de fusée, lequel peut être à combustible liquide ou solide. Par exemple, comment modéliser une fusée à combustible liquide bon marché ?
- Le modèle montre que Firework est une sous-classe de LiquidRocket impliquant que tous les artifices sont des fusées à combustible liquide, ce qui est faux.
- Le modèle montre une relation directe entre les réservations et les types d'artifices, bien qu'aucune relation n'existe dans le domaine de problèmes.
- La classe Reservation possède sa propre copie de city, qu'elle devrait obtenir par délégation à un objet Location.
- CheapRockets se compose d'objets Runnable, ce qui est tout simplement étrange.

Solution 7.2

L'intérêt de cet exercice est surtout de vous amener à réfléchir sur ce qui constitue une bonne classe. Voyez si votre définition considère les points suivants :

- Voici une description basique d'une classe : "Un ensemble nommé de champs contenant des valeurs de données, et de méthodes qui opèrent sur ces valeurs" [Flanagan 2005, p. 71].
- Une classe établit un ensemble de champs, c'est-à-dire les attributs d'un objet. Le type de chaque attribut peut être une classe, un type de données primitif, tel que boolean et int, ou une interface.
- Un concepteur de classes devrait être en mesure d'expliquer de quelle façon les attributs d'une classe sont liés.

- Une classe devrait remplir un objectif logique.
- Le nom d'une classe devrait refléter la signification de la classe, à la fois en tant que collection d'attributs et en ce qui concerne son comportement.
- Une classe doit supporter tous les comportements qu'elle définit, ainsi que ceux des super-classes et toutes les méthodes des interfaces qu'elle implémente — la décision de ne pas supporter une méthode d'une super-classe ou d'une interface peut occasionnellement se justifier.
- La relation d'une classe vis-à-vis de sa super-classe doit être justifiable.
- Le nom de chaque méthode d'une classe devrait constituer un bon commentaire de ce qu'elle accomplit.

Solution 7.3

Deux bonnes remarques sont que le résultat d'invoquer une opération peut dépendre de l'état de l'objet récepteur ou de la classe de l'objet récepteur. A d'autres moments, vous utilisez un nom imposé par quelqu'un d'autre.

Un exemple de méthode dont l'effet dépend de l'état de l'objet concerné apparaît au Chapitre 6, où la classe `MachineManager2` possède une méthode `stopMachine()`. Le résultat de l'appel de cette méthode dépend du driver qui est en place pour l'objet `MachineManager2`.

Lorsque le polymorphisme fait partie de la conception, l'effet d'invoquer une opération peut partiellement ou totalement dépendre de la classe de l'objet récepteur. Ce principe apparaît dans de nombreux patterns, surtout avec **FACTORY METHOD**, **STATE**, **STRATEGY**, **COMMAND** et **INTERPRETER**. Par exemple, les classes de stratégie d'une hiérarchie peuvent toutes implémenter une méthode `getRecommended()`, en utilisant différentes stratégies pour recommander un artifice particulier. Il est facile de comprendre que `getRecommended()` recommandera un artifice ; mais sans connaître la classe de l'objet qui reçoit l'appel de la méthode, il est impossible de connaître la stratégie sous-jacente qui sera utilisée.

Une troisième situation se produit lorsqu'une autre personne définit le nom. Suppossez que votre méthode agisse en tant que callback : vous avez redéfini la méthode `mouseDown()` de la classe `MouseListener`. Vous devez utiliser `mouseDown()` comme nom de méthode, même s'il n'indique rien sur l'intention de votre méthode.

Solution 7.4

Le code compile sans problème. L'accès est défini au niveau classe et non au niveau objet. Aussi un objet Firework peut-il accéder aux variables et méthodes privées d'un autre objet Firework, par exemple.

SINGLETON

Solution 8.1

Pour empêcher d'autres développeurs d'instancier votre classe, créez un seul constructeur avec un accès `private`. Notez que si vous créez d'autres constructeurs non privés, ou ne créez pas de constructeurs du tout, d'autres objets seront en mesure d'instancier votre classe.

Solution 8.2

Voici deux raisons de recourir à l'initialisation tardive, ou paresseuse (*lazy*) :

1. Vous pourriez ne pas disposer de suffisamment d'informations pour instancier un singleton au moment voulu. Par exemple, un singleton Factory pourrait être forcé d'attendre que les machines réelles établissent des canaux de communication entre elles.
2. Vous pourriez choisir d'initialiser tardivement un singleton qui requiert des ressources, telles qu'une connexion à une base de données, surtout s'il y a une possibilité que l'application conteneur ne requière pas le singleton durant une certaine session.

Solution 8.3

Votre solution devrait éliminer toute possibilité de confusion pouvant se produire lorsque deux threads appellent la méthode `recordWipMove()` à peu près au même moment :

```
public void recordWipMove() {  
    synchronized (classLock) {  
        wipMoves++;  
    }  
}
```

Est-il possible qu'un thread puisse devenir actif au beau milieu d'une opération d'incrémentation ? Absolument : toutes les machines n'ont pas qu'une seule instruction pouvant incrémenter une variable, et même celles qui sont dans ce cas ne peuvent pas garantir que le compilateur les utilisera dans toutes les situations. C'est une bonne stratégie que de restreindre l'accès aux données d'un singleton dans une application multithread. Voir *Concurrent Programming in Java™* [Lea 2000] pour plus d'informations sur de telles applications.

Solution 8.4

OurBiggestRocket:	Cette classe a un nom approprié. Vous devriez normalement modéliser des propriétés, telles que "biggest" par exemple, par des attributs et non par des noms de classes. Si un développeur <i>devait</i> gérer cette classe, ce serait peut-être en tant que singleton.
topSalesAssociate:	Cette classe présente le même problème que OurBiggestRocket.
Math:	C'est une classe <i>utilitaire</i> , avec uniquement des méthodes statiques et <i>aucune</i> instance. Ce n'est pas un singleton. Notez qu'elle possède toutefois un constructeur privé.
System:	C'est également une classe utilitaire.
PrintStream:	Bien que l'objet System.out soit un objet PrintStream avec des responsabilités uniques, il ne constitue pas une instance unique de PrintStream, qui n'est pas un singleton.
PrintSpooler:	Un PrintSpooler est associé à une imprimante ou à quelques imprimantes. Il est peu vraisemblable que ce soit un singleton.
PrinterManager:	Chez Oozinoz, il y a plusieurs imprimantes et vous pouvez rechercher leurs adresses respectives par l'intermédiaire du singleton PrinterManager.

OBSERVER

Solution 9.1

Une solution possible est :

```
public JSlider slider() {
    if (slider == null) {
        slider = new JSlider();
        sliderMax = slider.getMaximum();
        sliderMin = slider.getMinimum();
        slider.addChangeListener(this);
        slider.setValue(slider.getMinimum());
    }
    return slider;
}
public void stateChanged(ChangeEvent e) {
    double val = slider.getValue();
```

```

        double tp = (val - sliderMin) / (sliderMax - sliderMin);
        burnPanel().setTPeak(tp);
        thrustPanel().setTPeak(tp);
        valueLabel().setText(Format.formatToNPlaces(tp, 2));
    }
}

```

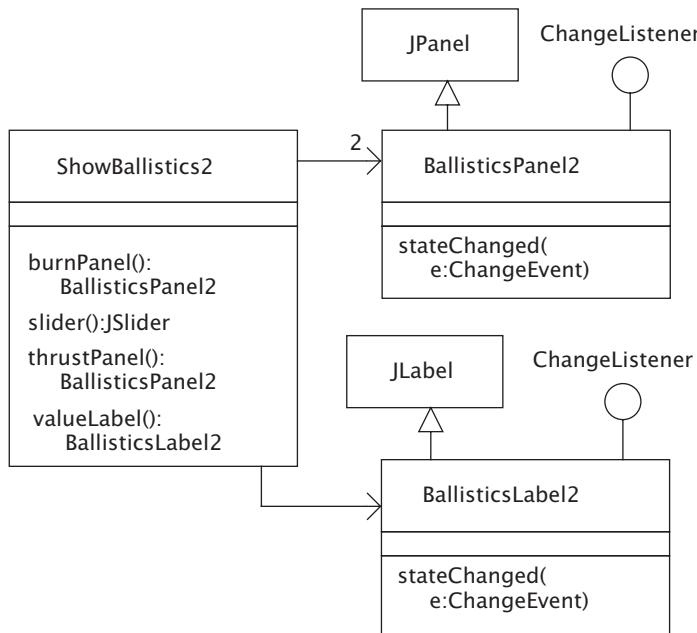
Ce code suppose qu'une classe auxiliaire Format existe pour formater le champ de valeur. Vous pourriez avoir utilisé à la place une expression telle que `" "+tp` ou `Double.toString(tp)` — l'emploi d'un nombre de chiffres constant produit une animation plus fluide.

Solution 9.2

Une solution est montrée à la Figure B.7. Pour permettre à un champ de valeur de s'enregistrer pour être notifié des événements de curseur, la conception dans la Figure B.7 crée une sous-classe `JLabel` qui implémente `ChangeListener`.

Figure B.7

Dans cette conception, les composants qui dépendent du curseur implémentent `ChangeListener` pour pouvoir s'enregistrer et être notifiés des événements de déplacement du curseur.



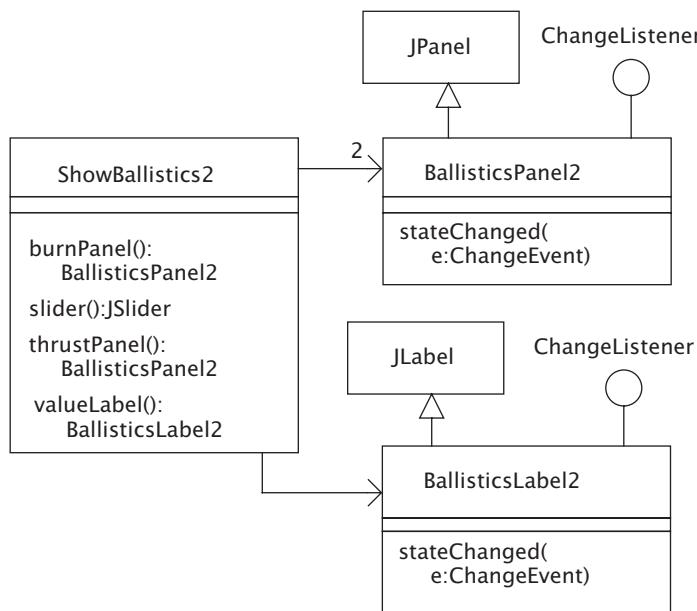
La nouvelle conception permet aux composants dépendants du curseur de s'enregistrer pour être notifiés et de s'actualiser ainsi eux-mêmes. C'est une amélioration discutable, mais nous restructurerons à nouveau la conception vers une architecture Modèle-Vue-Contrôleur.

Solution 9.3

La Figure B.8 illustre une solution.

Figure B.8

Cette conception prévoit que l'application surveille le curseur. Le champ de valeur et les panneaux d'affichage restent attentifs aux changements d'un objet contenant la valeur tPeak.



Un objet Tpeak qui contient une valeur de temps crée joue un rôle central dans cette conception. L'application `ShowBallistics3` crée l'objet Tpeak, et le curseur l'actualise lorsque sa position change. Les composants d'affichage (le champ de valeur et les panneaux de tracé) restent "à l'écoute" de l'objet Tpeak en s'enregistrant en tant qu'observateurs de l'objet.

Solution 9.4

Voici une solution possible :

```

package app.observer.ballistics3;
import javax.swing.*;
import java.util.*;
public class BallisticsLabel extends JLabel
    implements Observer {
    public BallisticsLabel(Tpeak tPeak) {
        tPeak.addObserver(this);
    }
    public void update(Observable o, Object arg) {
        setText(" " + ((Tpeak) o).getValue());
        repaint();
    }
}
  
```

Solution 9.5

Voici une solution possible :

```
protected JSlider slider() {
    if (slider == null) {
        slider = new JSlider();
        sliderMax = slider.getMaximum();
        sliderMin = slider.getMinimum();
        slider.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                if (sliderMax == sliderMin) return;
                tPeak.setValue(
                    (slider.getValue() - sliderMin)
                    / (sliderMax - sliderMin));
            }
        });
        slider.setValue(slider.getMinimum());
    }
    return slider;
}
```

Solution 9.6

La Figure B.9 illustre le flux d'appels qui se produit lorsqu'un utilisateur déplace le curseur de l'application de calculs balistiques.

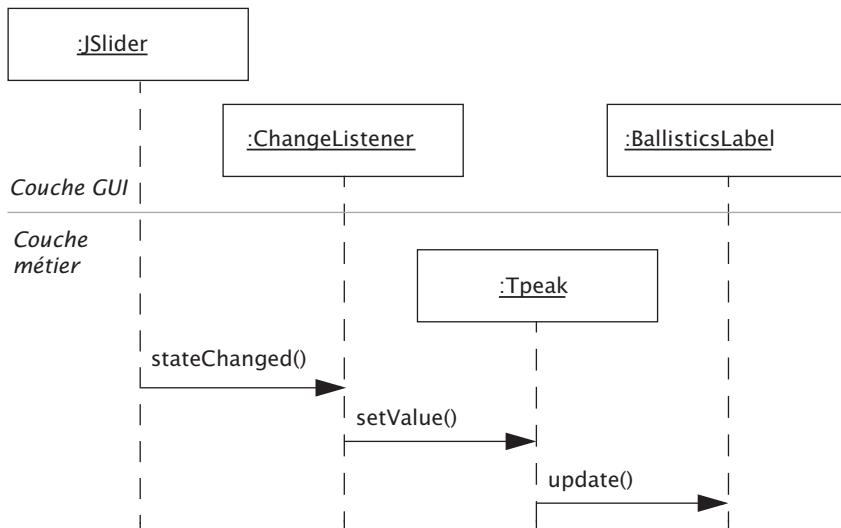


Figure B.9

La conception MVC force un cheminement des messages par une couche métier.

Solution 9.7

Votre diagramme devrait ressembler à l'exemple de la Figure B.10. Notez que vous pouvez appliquer la même conception avec `Observer` et `Observable`. La clé de cette conception réside dans le fait que la classe `Tpeak` intéressante se rend elle-même observable en gérant un objet avec des capacités d'écoute.

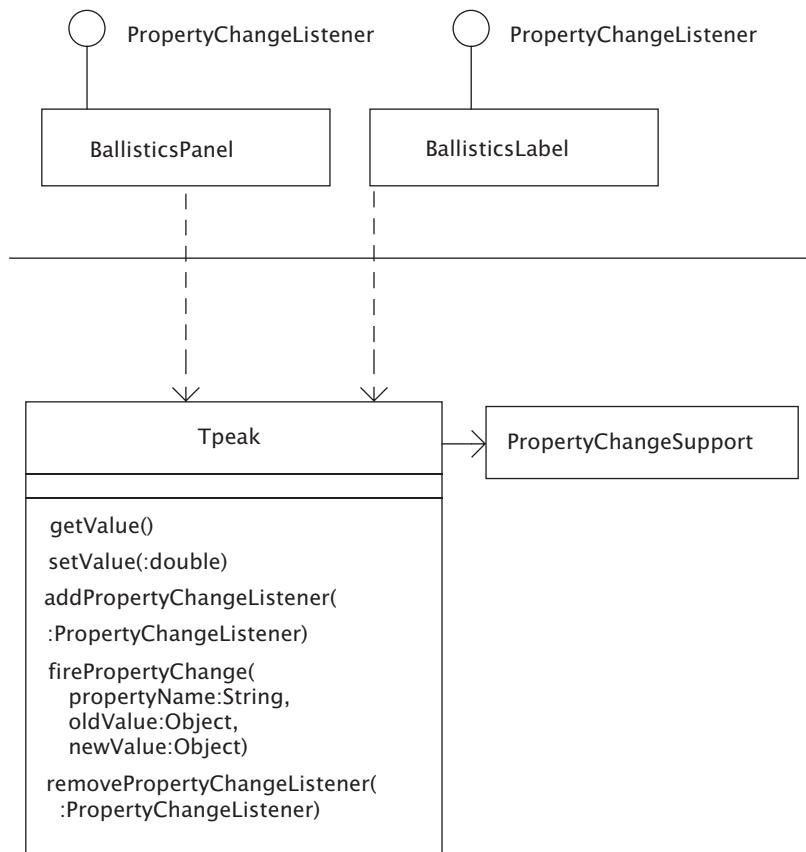


Figure B.10

La classe `Tpeak` peut ajouter des fonctionnalités d'écoute en déléguant les appels orientés écoute à un objet `PropertyChangeSupport`.

MEDIATOR

Solution 10.1

La Figure B.11 illustre une solution.

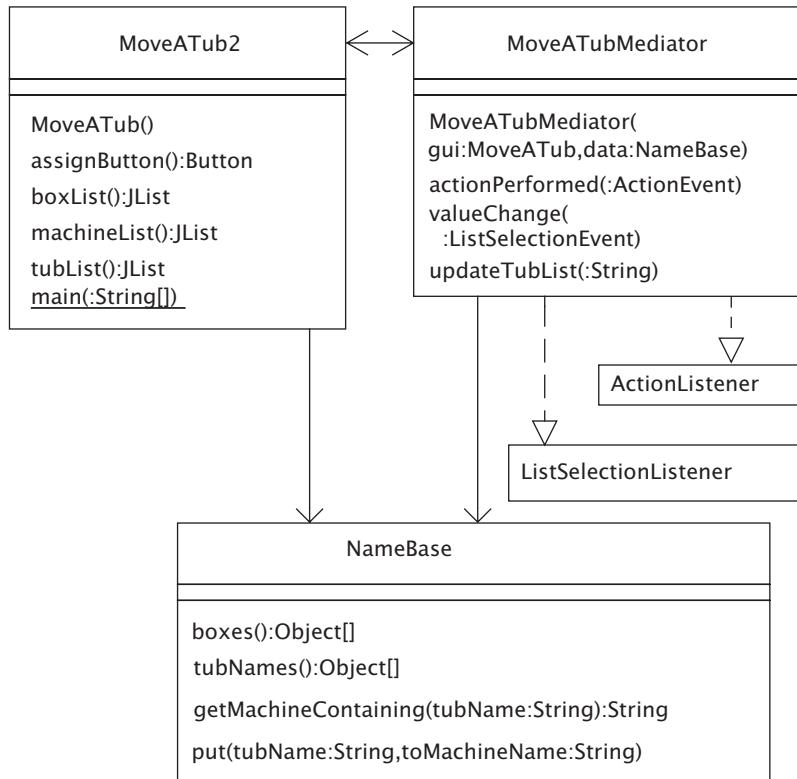


Figure B.11

La classe *MoveATub* gère la construction de composants et la classe *MoveATubMediator* gère les événements.

Dans cette conception, la classe de médiation révèle ce qui est cité dans Fowler et al. [1999] comme étant un symptôme de "Feature Envy", où elle semble plus intéressée par la classe de GUI que par elle-même, comme le montre la méthode `valueChanged()` :

```

public void valueChanged(ListSelectionEvent e) {
    // ...
  
```

```

        gui.assignButton().setEnabled(
            ! gui.tubList().isSelectionEmpty()
            && ! gui.machineList().isSelectionEmpty());
    }
}

```

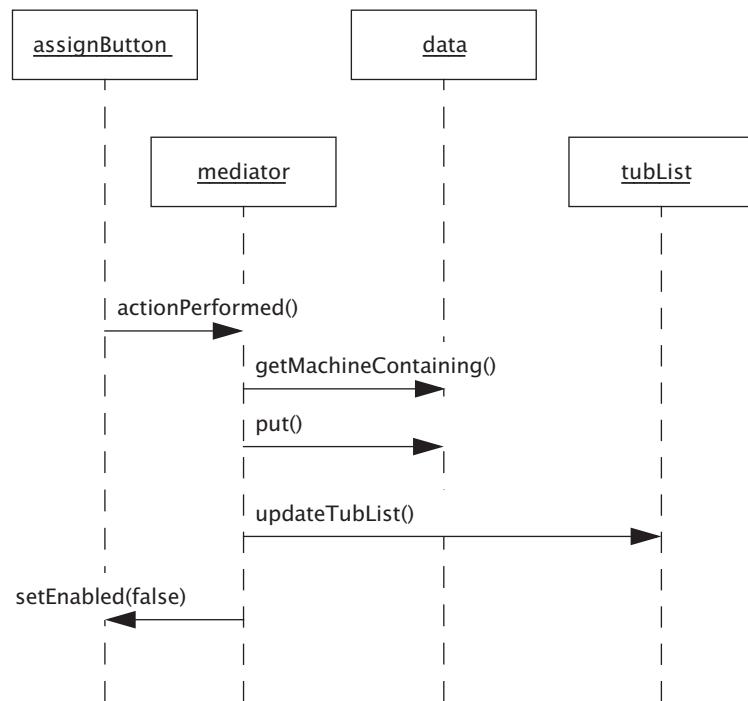
Le rejet par certains développeurs de cet aspect "désir de fonctionnalité" les écarte de ce genre de conception. Vous pourrez toutefois éprouver le besoin d'avoir une classe pour la construction et le placement d'un composant de GUI et une classe séparée pour gérer l'interaction avec le composant.

Solution 10.2

La Figure B.12 illustre une solution possible.

Figure B.12

Ce diagramme illustre le rôle central du médiateur.



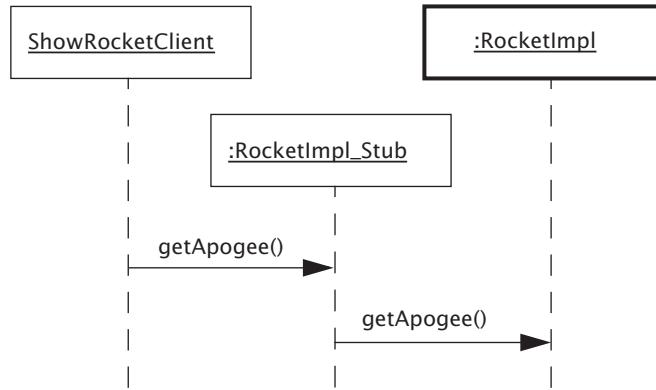
La solution fournie met en valeur le rôle du médiateur en tant que dispatcher, recevant un événement et assumant la responsabilité d'actualiser tous les objets concernés.

Solution 10.3

La Figure B.13 présente un diagramme objet actualisé.

Figure B.13

Deux machines pensent qu'elles contiennent un bac T308. Le modèle objet accepte une situation que ni une table relationnelle ni la réalité n'accepteraient — les traits épais soulignent la situation problématique.



Le problème que le code du développeur introduit est que la machine StarPress-2402 pense toujours qu'elle possède un bac T308. Dans une table relationnelle, le changement des attributs de machine d'une ligne retire automatiquement le bac de la machine précédente. Cette suppression automatique ne se produit pas lorsque la relation est dispersée à travers un modèle objet réparti. La modélisation correcte de la relation bac/machine nécessite une logique spéciale que vous pouvez déplacer vers un objet médiateur distinct.

Solution 10.4

Le code complet pour la classe TubMediator devrait ressembler à l'exemple suivant :

```

package com.oozinoz.machine;
import java.util.*;
public class TubMediator {
    protected Map tubToMachine = new HashMap();
    public Machine getMachine(Tub t) {
        return (Machine) tubToMachine.get(t);
    }
    public Set getTubs(Machine m) {
        Set set = new HashSet();
    }
}
  
```

```
Iterator i = tubToMachine.entrySet().iterator();
while (i.hasNext()) {
    Map.Entry e = (Map.Entry) i.next();
    if (e.getValue().equals(m))
        set.add(e.getKey());
}
return set;
}
public void set(Tub t, Machine m) {
    tubToMachine.put(t, m);
}
}
```

Solution 10.5

- Le pattern FACADE peut aider à la restructuration d'une grande application.
- Le pattern BRIDGE peut placer les opérations abstraites dans une interface.
- Le pattern OBSERVER peut intervenir pour restructurer du code pour supporter une architecture MVC.
- Le pattern FLYWEIGHT extrait la partie immuable d'un objet pour qu'elle puisse être partagée.
- Le pattern BUILDER place la logique de construction d'un objet en dehors de la classe à instancier.
- Le pattern FACTORY METHOD permet de réduire les responsabilités d'une hiérarchie de classes en plaçant un aspect des comportements dans une hiérarchie parallèle.
- Les patterns STATE et STRATEGY permettent de placer les comportements spécifiques à un état ou à une stratégie dans des classes distinctes.

PROXY

Solution 11.1

Voici une solution possible :

```
public int getIconHeight() {
    return current.getIconHeight();
}
public int getIconWidth() {
    return current.getIconWidth();
}
```

```
public synchronized void paintIcon(
    Component c, Graphics g, int x, int y) {
    current.paintIcon(c, g, x, y);
}
```

Solution 11.2

Voici quelques-uns des problèmes que présente cette conception :

- La transmission d'un sous-ensemble d'appels seulement à un objet `ImageIcon` sous-jacent est périlleuse. La classe `ImageIconProxy` hérite d'une dizaine de champs et d'au moins 25 méthodes de la classe `ImageIcon`. Pour être un véritable proxy, l'objet `ImageIconProxy` devrait transmettre la totalité ou la plupart des appels. Une retransmission détaillée nécessiterait beaucoup de méthodes potentiellement fautives, et ce code demanderait de la maintenance supplémentaire à mesure que la classe `ImageIcon` et ses super-classes changeraient dans le temps.
- Vous pouvez vous demander si l'image "absente" et l'image souhaitée se trouvent au bon endroit dans la conception. Il pourrait être plus logique que les images soient passées plutôt que de rendre la classe responsable de leur obtention.

Solution 11.3

La méthode `load()` définit comme image "Loading...", alors que la méthode `run()`, qui s'exécute dans un thread distinct, charge l'image désirée :

```
public void load(JFrame callbackFrame) {
    this.callbackFrame = callbackFrame;
    setImage(LOADING.getImage());
    callbackFrame.repaint();
    new Thread(this).start();
}
public void run() {
    setImage(new ImageIcon(
        ClassLoader.getSystemResource(filename))
        .getImage());
    callbackFrame.pack();
}
```

Solution 11.4

Comme le montre le diagramme de classes, un constructeur `RocketImpl` accepte une valeur de prix et d'apogée.

```
Rocket biggie = new rocketImpl(29.95, 820);
```

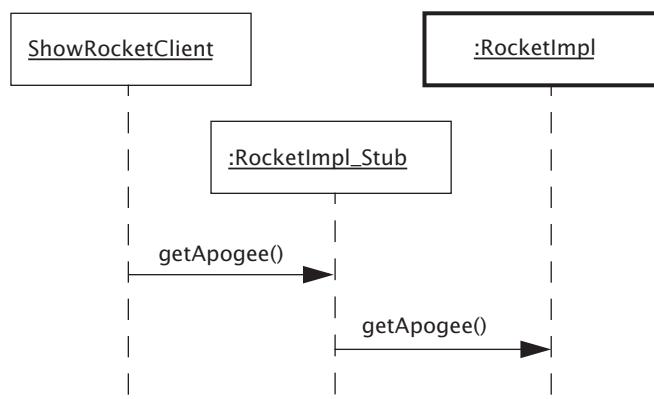
Vous pourriez déclarer `biggie` de type `RocketImpl`. Toutefois, l'important à propos de `biggie` est qu'il remplisse l'objectif de l'interface `Rocket` qu'un client rechercherait.

Solution 11.5

Le diagramme complété devrait ressembler à l'illustration de la Figure B.14. Une autre décision légitime, toutefois moins explicite, pourrait déclarer les deux récepteurs de `getApogee()` comme étant de type `Rocket`. En fait, les programmes serveur et client se réfèrent tous deux à ces objets en tant qu'instances de l'interface `Rocket`.

Figure B.14

Un proxy transmet les appels d'un client de sorte que l'objet distant apparaisse local au client.



CHAIN OF RESPONSABILITY

Solution 12.1

Voici quelques-uns des inconvénients possibles de la conception CHAIN OF RESPONSABILITY utilisée par Oozinoz pour trouver l'ingénieur responsable d'une machine :

- Nous n'avons pas spécifié comment la chaîne sera mise en place pour que les machines connaissent leur parent. Dans la pratique, il peut se révéler difficile de garantir que les parents ne soient jamais null.
- Il est concevable que le processus de recherche d'un parent entre dans une boucle infinie, selon la façon dont les parents sont implémentés.

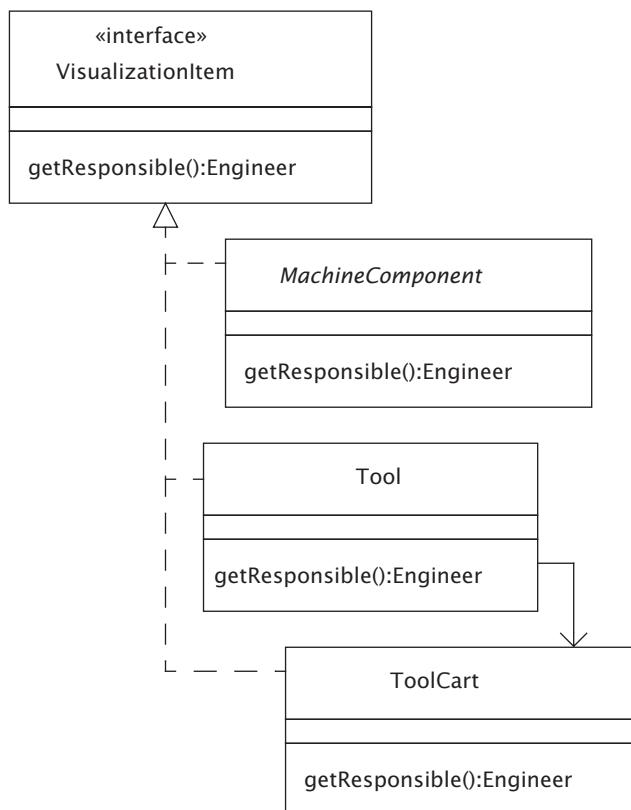
- Tous les objets n'offrent pas tous les comportements impliqués par ces nouvelles méthodes. Par exemple, l'élément du niveau supérieur n'a pas de parent.
- La conception présente est insuffisante quant aux détails afférents à la façon dont le système sait quels ingénieurs sont actuellement présents dans l'usine et disponibles. On ne sait pas clairement quelle devrait être, en temps réel, la portée de cette responsabilité.

Solution 12.2

Votre diagramme devrait ressembler à l'exemple présenté dans la Figure B.15.

Figure B.15

Chaque objet *VisualizationItem* peut indiquer son ingénieur responsable. En interne, un tel objet peut transmettre la requête à un autre objet parent.



Avec cette conception, n'importe quel client de n'importe quel élément simulé peut simplement demander à l'élément son ingénieur responsable. Cette approche libère le client de la tâche de détermination des objets capables de comprendre la responsabilité et place cette charge au niveau des objets qui implémentent l'interface `VisualizationItem`.

Solution 12.3

- A. Un objet `MachineComponent` peut avoir un responsable assigné. Si ce n'est pas le cas, il passe la requête à son parent :

```
public Engineer getResponsible() {  
    if (responsible != null)  
        return responsible;  
    if (parent != null)  
        return parent.getResponsible();  
    return null;  
}
```

- B. Le code de `Tool.Responsible` reflète la règle stipulant que "les outils sont toujours assignés aux chariots d'outils" :

```
public Engineer getResponsible() {  
    return toolCart.getResponsible();  
}
```

- C. Le code de `ToolCart` reflète la règle stipulant que "les chariots d'outils ont un ingénieur responsable" :

```
public Engineer getResponsible() {  
    return responsible;  
}
```

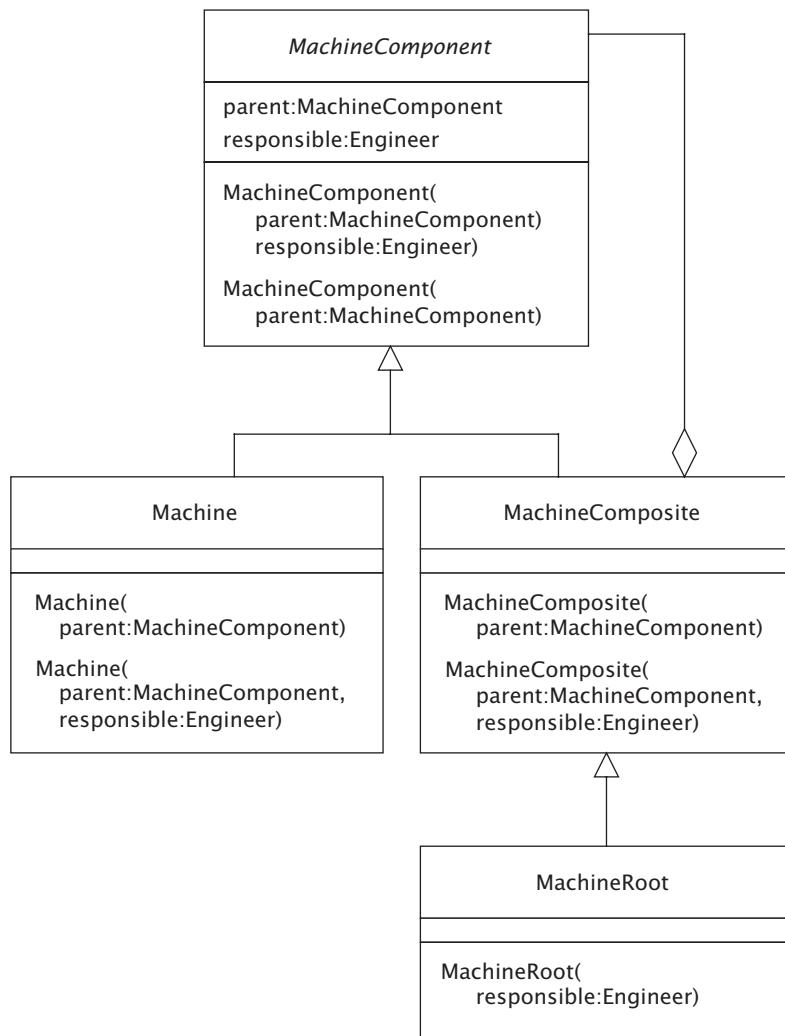
Solution 12.4

Votre solution devrait ressembler au diagramme de la Figure B.16.

Vos constructeurs devraient autoriser les objets `Machine` et `MachineComposite` à être instanciés avec ou sans responsable assigné. Chaque fois qu'un objet `MachineComponent` ne possède pas d'ingénieur assigné, il peut obtenir un ingénieur de son parent.

Figure B.16

Les constructeurs dans la hiérarchie MachineComponent supportent les règles stipulant qu'un objet MachineRoot doit avoir un ingénieur responsable et que chaque objet Machine-Component, excepté l'objet racine, doit avoir un parent.



Solution 12.5

Le pattern CHAIN OF RESPONSABILITY pourrait s'appliquer aux objets non composites dans les situations suivantes :

- Une chaîne d'ingénieurs de permanence suivent une rotation standard. Si l'ingénieur de permanence principal ne répond pas à un appel par pageur du service de

production dans un certain intervalle de temps, le système de notification appelle le prochain ingénieur dans la chaîne.

- Des utilisateurs saisissent des informations, telles que la date d'un événement, et une chaîne d'analyseurs syntaxiques tentent à tour de rôle de décoder le texte saisi.

FLYWEIGHT

Solution 13.1

Un argument en faveur de l'immuabilité des chaînes. Dans la pratique, les chaînes sont fréquemment partagées entre plusieurs clients, ce qui constitue l'essentiel des défauts qui apparaissent lorsqu'un client a accidentellement un impact sur un autre. Par exemple, une méthode qui retourne le nom d'un client sous forme d'une chaîne conservera généralement une référence au nom. Si le client convertit la chaîne en lettre majuscules pour l'utiliser dans une table indexée par hachage, le nom de l'objet Customer changera également, si ce n'est pour l'immuabilité des chaînes. Dans Java, vous pouvez produire une version d'une chaîne en majuscules, mais il faut que cela soit un nouvel objet, et non une version modifiée de la chaîne initiale. L'immuabilité des chaînes permet de les partager en toute sécurité. De plus, cela aide le système à éviter certains risques pour la sécurité.

Un argument contre l'immuabilité des chaînes. L'immuabilité des chaînes nous évite de commettre certaines erreurs mais le prix à payer est lourd. Premièrement, le développeur est privé de la possibilité de modifier une chaîne, indépendamment de la façon dont il pourrait justifier ce besoin. Deuxièmement, l'ajout de certaines règles spéciales à un langage le rend plus difficile à apprendre et à utiliser. Java est bien plus difficile à apprendre que le langage Smalltalk, qui est aussi puissant. Finalement, aucun langage informatique ne peut m'empêcher de commettre des erreurs. Je préfère pouvoir apprendre un langage rapidement et avoir ainsi le temps d'apprendre à mettre en place et utiliser un environnement de test.

Solution 13.2

Vous pouvez déplacer les aspects immuables de Substance — nom, symbole et poids atomique inclus — vers une classe Chemical, comme le montre la Figure B.17.

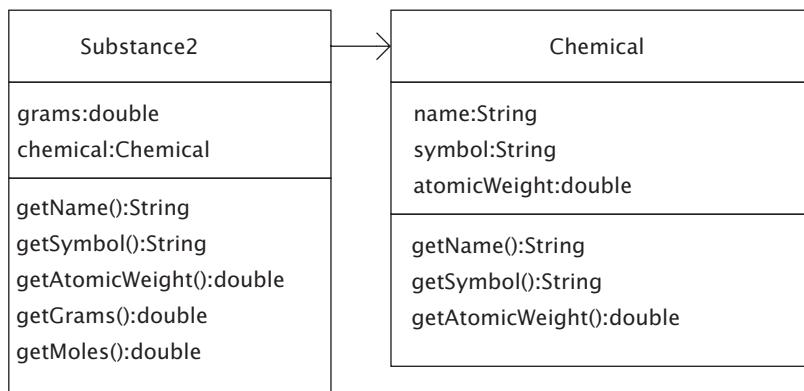


Figure B.17

Ce diagramme montre la portion immuable de la classe Substance placée dans une classe Chemical.

La classe **Substance2** conserve maintenant une référence à un objet **Chemical**. Par conséquent, la classe **Substance2** peut toujours offrir les mêmes méthodes accesseurs que la classe **Substance** originale. En interne, ces accesseurs s'appuient sur la classe **Chemical**, comme le montrent les méthodes de **Substance2** suivantes :

```
public double getAtomicWeight() {  
    return chemical.getAtomicWeight();  
}  
public double getGrams() {  
    return grams;  
}  
public double getMoles() {  
    return grams / getAtomicWeight();  
}
```

Solution 13.3

Un moyen qui ne fonctionnera pas est de déclarer `private` le constructeur de `Chemical`. Cela empêcherait la classe `ChemicalFactory` d'instancier la classe `Chemical`.

Pour contribuer à empêcher les développeurs d'instancier eux-mêmes la classe `Chemical`, vous pouvez placer les classes `Chemical` et `ChemicalFactory` dans le même package et offrir un accès par défaut ("package") au constructeur de la classe `Chemical`.

Solution 13.4

L'approche par classe imbriquée est bien plus complexe mais plus complète pour s'assurer que seule la classe `ChemicalFactory2` peut instancier de nouveaux objets flyweight. Le code résultant devrait ressembler à l'exemple suivant :

```
package com.oozinoz.chemical2;
import java.util.*;
public class ChemicalFactory2 {
    private static Map chemicals = new HashMap();
    class ChemicalImpl implements Chemical {
        private String name;
        private String symbol;
        private double atomicWeight;
        ChemicalImpl(
            String name,
            String symbol,
            double atomicWeight) {
            this.name = name;
            this.symbol = symbol;
            this.atomicWeight = atomicWeight;
        }
        public String getName() {
            return name;
        }
        public String getSymbol() {
            return symbol;
        }
        public double getAtomicWeight() {
            return atomicWeight;
        }
        public String toString() {
            return name + "(" + symbol + ")[ " +
                atomicWeight + "]";
        }
    }
    static {
        ChemicalFactory2 factory = new ChemicalFactory2();
```

```
chemicals.put("carbon",
    factory.new ChemicalImpl("Carbon", "C", 12));
chemicals.put("sulfur",
    factory.new ChemicalImpl("Sulfur", "S", 32));
chemicals.put("saltpeter",
    factory.new ChemicalImpl(
        "Saltpeter", "KN03", 101));
//...
}
public static Chemical getChemical(String name) {
    return (Chemical) chemicals.get(
        name.toLowerCase());
}
}
```

Ce code répond aux trois exercices de la façon suivante :

1. La classe `ChemicalImpl` imbriquée devrait être privée pour que seule la classe `ChemicalFactory2` puisse l'utiliser. Notez que l'accès à la classe imbriquée devrait être de niveau package ou public pour que la classe conteneur puisse l'instancier. Même si vous déclariez le constructeur public, aucune autre classe ne pourrait utiliser le constructeur si la classe imbriquée elle-même est déclarée `private`.
2. Le constructeur de `ChemicalFactory2` utilise un initialisateur statique pour garantir que la classe ne construira qu'une seule fois la liste de substances chimiques.
3. La méthode `getChemical()` devrait rechercher les substances chimiques par nom dans la table indexée par hachage. L'exemple de code stocke et recherche les substances en utilisant la version en minuscules de leur nom.

Introduction à la construction

Solution 14.1

Voici quelques-unes des règles spéciales concernant les constructeurs :

- Si vous ne fournissez pas de constructeur pour une classe, Java en fournira un par défaut.
- Les noms de constructeurs doivent correspondre aux noms de leurs classes respectives — pour cette raison, ils commencent généralement par une capitale, à la différence de la plupart des noms de méthodes.
- Les constructeurs peuvent invoquer d'autres constructeurs avec `this()` et `super()` tant que cette invocation représente la première instruction dans le constructeur.

- Le "résultat" de l'action d'un constructeur est une instance de la classe, alors que le type de retour d'une méthode ordinaire peut être n'importe quoi.
- Vous pouvez utiliser new ou la réflexion pour invoquer un constructeur.

Solution 14.2

Le code suivant échouera à la compilation une fois placé dans Fuse.java et QuickFuse.java.

```
package app.construction;
public class Fuse {
    private String name;
    public Fuse(String name) { this.name = name; }
}
```

et

```
package app.construction;
public class QuickFuse extends Fuse { }
```

Le compilateur générera un message d'erreur dont la teneur sera à peu près la suivante :

Constructeur Fuse() super implicite indéfini pour un constructeur par défaut. Constructeur explicite obligatoire.

Cette erreur se produit lorsque le compilateur rencontre la classe QuickFuse et fournit un constructeur par défaut. Celui-ci n'attend pas d'argument et invoque, à nouveau par défaut, le constructeur de la super-classe sans argument. Toutefois, la présence d'un constructeur Fuse() qui accepte un paramètre String signifie que le compilateur ne fournira plus de constructeur par défaut pour Fuse. Le constructeur par défaut de QuickFuse ne peut alors plus invoquer le constructeur de la super-classe sans argument car il n'existe plus.

Solution 14.3

Le programme affiche :

```
java.awt.Point[x=3,y=4]
```

Le programme a réussi à trouver un constructeur acceptant deux arguments et a créé un nouveau point avec les arguments donnés.

BUILDER

Solution 15.1

Une façon de rendre l'analyseur plus flexible est de lui permettre d'accepter plusieurs espaces après une virgule. Pour cela, changez la construction de l'appel de `split()` de la manière suivante :

```
s.split(", *");
```

Ou bien, au lieu d'accepter des espaces après une virgule, vous pouvez autoriser n'importe quel type d'espace en initialisant l'objet `Regex` comme suit :

```
s.split(", \s*)")
```

La combinaison de caractères `\s` signifie la "catégorie" de caractères blancs dans les expressions régulières. Notez que ces solutions supposent qu'il n'y aura pas de virgule insérée dans les champs.

Plutôt que de rendre l'analyseur plus souple, vous pouvez remettre en question l'approche. En particulier, vous pouvez solliciter des agences de voyages qu'elles commencent à envoyer les réservations au format XML. Vous pouvez établir un ensemble de balises de marquage à utiliser et les lire au moyen d'un analyseur XML.

Solution 15.2

La méthode `build()` de `UnforgivingBuilder` génère une exception si un attribut est invalide et retourne sinon un objet `Reservation` valide. Voici une implémentation :

```
public Reservation build() throws BuilderException {
    if (date == null)
        throw new BuilderException("Valid date not found");
    if (city == null)
        throw new BuilderException("Valid city not found");
    if (headcount < MINHEAD)
        throw new BuilderException(
            "Minimum headcount is " + MINHEAD);
    if (dollarsPerHead.times(headcount)
        .isLessThan(MINTOTAL))
        throw new BuilderException(
            "Minimum total cost is " + MINTOTAL);
    return new Reservation(
        date,
        headcount,
        city,
        dollarsPerHead,
        hasSite);
}
```

Le code vérifie que les valeurs de date et de ville sont définies et que les valeurs de nombre de personnes et de prix par tête sont acceptables. La super-classe `ReservationBuilder` définit les constantes `MINHEAD` et `MINTOTAL`.

Si le constructeur ne rencontre pas de problèmes, il retourne un objet `Reservation` valide.

Solution 15.3

Comme vu précédemment, le code doit générer une exception si la réservation omet de spécifier une ville ou une date car il n'y a aucun moyen de deviner ces valeurs. Pour ce qui est de l'omission des valeurs de nombre de personne et de prix par tête, notez les points suivants :

- Si la requête de réservation ne spécifie ni le nombre de personnes ni le prix par tête, définissez la quantité de personnes à la valeur minimale, et assignez au prix par tête le résultat de la division du prix total acceptable par la quantité minimale de personnes.
- Si le nombre de personnes est omis mais pas le prix par tête, définissez la quantité de personnes à la valeur minimale tout en veillant à ce qu'elle soit suffisante pour atteindre la recette totale minimale acceptable pour un spectacle.
- Si le nombre de personnes est indiqué mais pas le prix par tête, définissez celui-ci suffisamment haut pour générer la recette minimale.

Solution 15.4

Voici une solution possible :

```
public Reservation build() throws BuilderException {  
    boolean noHeadcount = (headcount == 0);  
    boolean noDollarsPerHead = (dollarsPerHead.isZero());  
    if (noHeadcount && noDollarsPerHead) {  
        headcount = MINHEAD;  
        dollarsPerHead = sufficientDollars(headcount);  
    } else if (noHeadcount) {  
        headcount = (int) Math.ceil(  
            MINTOTAL.dividedBy(dollarsPerHead));  
        headcount = Math.max(headcount, MINHEAD);  
    } else if (noDollarsPerHead) {  
        dollarsPerHead = sufficientDollars(headcount);  
    }  
}
```

```
    check();
    return new Reservation(
        date,
        headcount,
        city,
        dollarsPerHead,
        hasSite);
}
```

Ce code s'appuie sur une méthode `check()` qui est similaire à la méthode `build()` de la classe `InforgivingBuilder` :

```
protected void check() throws BuilderException {
    if (date == null)
        throw new BuilderException("Valid date not found");
    if (city == null)
        throw new BuilderException("Valid city not found");
    if (headcount < MINHEAD)
        throw new BuilderException(
            "Minimum headcount is " + MINHEAD);
    if (dollarsPerHead.times(headcount)
        .isLessThan(MINTOTAL))
        throw new BuilderException(
            "Minimum total cost is " + MINTOTAL);
}
```

FACTORY METHOD

Solution 16.1

Une bonne réponse serait peut-être de dire que vous n'avez pas besoin d'identifier la classe sous-jacente à l'objet retourné par une méthode `iterator()`. Ce qui est important, c'est que vous connaissiez l'interface supportée par l'itérateur, ce qui vous permet d'énumérer les éléments d'une collection. Toutefois, si vous *devez* connaître la classe, vous pouvez afficher son nom avec une ligne d'instruction du genre :

```
System.out.println(iter.getClass().getName());
```

Cette instruction affiche :

```
java.util.AbstractList$Iter
```

La classe `Iter` est une classe interne de `AbstractList`. Vous ne devriez probablement jamais voir cette classe en travaillant avec Java.

Solution 16.2

Il y a de nombreuses réponses possibles, mais `toString()` est probablement la méthode la plus fréquemment utilisée pour créer un nouvel objet. Par exemple, le code suivant crée un nouvel objet `String` :

```
String s = new Date().toString();
```

La création de chaînes se produit souvent en coulisses. Considérez la ligne suivante :

```
System.out.println(new Date());
```

Ce code crée un objet `String` à partir de l'objet `Date` par l'intermédiaire de la méthode `toString()` de l'objet `Date`.

Une autre méthode couramment utilisée pour créer un nouvel objet est `clone()`, une méthode qui retourne généralement une copie de l'objet récepteur.

Solution 16.3

L'objectif du pattern FACTORY METHOD est de permettre à un fournisseur d'objets de déterminer quelle classe instancier lors de la création d'un objet. Par comparaison, les clients de `BorderFactory` savent exactement quels types d'objets ils obtiennent. Le pattern appliqué avec `BorderFactory` est FLYWEIGHT dans ce sens que `BorderFactory` emploie le partage pour supporter efficacement un grand nombre de bordures. La classe `BorderFactory` dispense les clients de devoir gérer la réutilisabilité des objets alors que FACTORY METHOD évite que les clients aient à savoir quelle classe instancier.

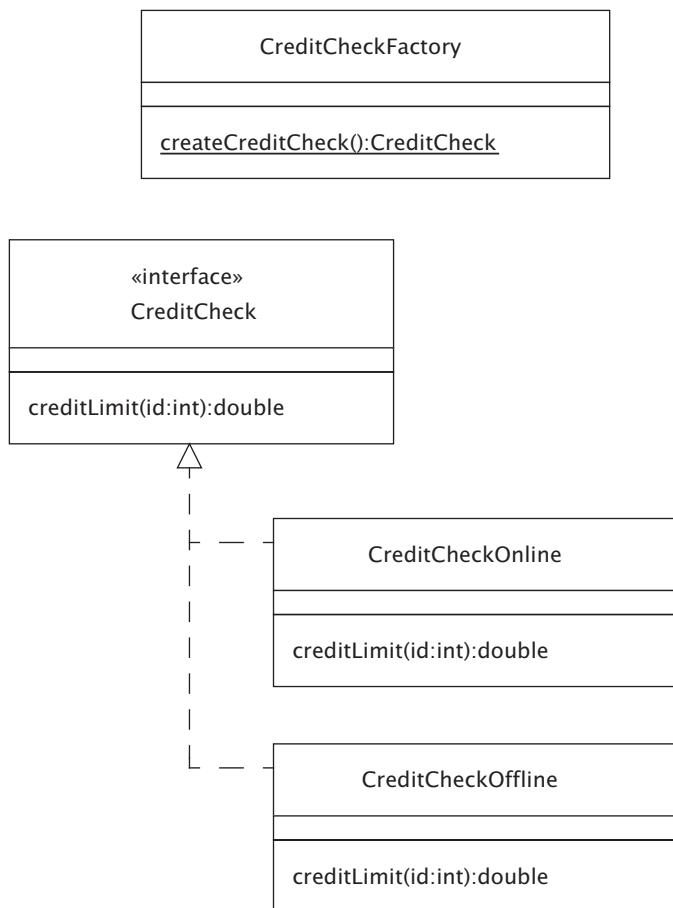
Solution 16.4

La Figure B.18 montre que les deux classes de vérification de limite de crédit implémentent l'interface `CreditCheck`. La classe de factory fournit une méthode qui retourne un objet `CreditCheck`. Le client qui appelle `createCreditCheck()` ne sait pas précisément quelle est la classe des objets qu'il obtient.

La méthode `createCreditCheck()` est statique. Aussi les clients n'ont-ils pas besoin d'instancier la classe `CreditCheckFactory` pour obtenir un objet `CreditCheck`. Vous pouvez définir cette classe abstraite ou prévoir un constructeur privé si vous voulez empêcher activement d'autres développeurs de l'instancier.

Figure B.18

Deux classes implémentent l'interface CreditCheck. Le choix de la classe à instancier dépend du fournisseur de services plutôt que du client concerné par la vérification de crédit.



Solution 16.5

En admettant que la méthode `isAgencyUp()` reflète précisément la réalité, le code pour `createCreditCheck()` est simple :

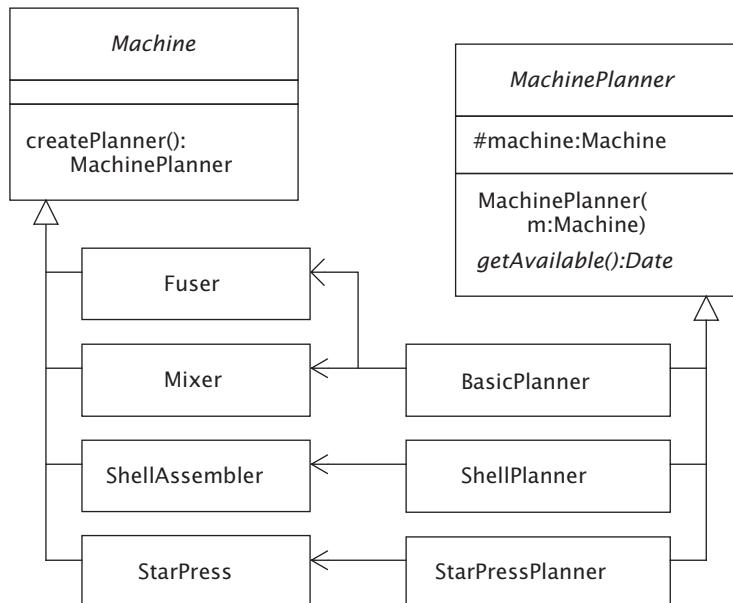
```
public static CreditCheck createCreditCheck() {
    if (isAgencyUp()) return new CreditCheckOnline();
    return new CreditCheckOffline();
}
```

Solution 16.6

La Figure B.19 illustre un diagramme acceptable pour la hiérarchie parallèle Machine/MachinePlanner.

Figure B.19

La logique de planification se trouve maintenant dans une hiérarchie distincte. Chaque sous-classe de Machine sait quel planificateur instancier en réponse à un appel de createPlanner().



Ce diagramme montre que les sous-classes de `MachinePlanner` doivent implémenter la méthode `getAvailable()`. Il indique aussi que les classes de la hiérarchie `MachinePlanner` acceptent un objet `Machine` dans leur constructeur. Cela permet au planificateur d'interroger l'objet bénéficiaire de la planification, relativement à des critères tels que l'emplacement de la machine et la quantité de produits qu'elle traite actuellement.

Solution 16.7

Une méthode `createPlanner()` pour la classe `Machine` pourrait ressembler à l'exemple suivant :

```

public MachinePlanner createPlanner() {
    return new BasicPlanner(this);
}
  
```

Les classes Fuser et Mixer peuvent compter sur le fait d'hériter cette méthode, alors que les classes ShellAssembler et StarPress devront la redéfinir. Pour la classe StarPress, la méthode `createPlanner()` pourrait se présenter comme suit :

```
public MachinePlanner createPlanner() {  
    return new StarPressPlanner(this);  
}
```

Ces méthodes illustrent le pattern FACTORY METHOD. Lorsque nous avons besoin d'un objet de planification, nous appelons la méthode `createPlanner()` de la machine concernée par la planification. Le planificateur spécifique que nous obtenons dépend de la machine.

ABSTRACT FACTORY

Solution 17.1

Voici une solution possible :

```
public class BetaUI extends UI {  
    public BetaUI () {  
        Font oldFont = getFont();  
        font = new Font(  
            oldFont.getName(),  
            oldFont.getStyle() | Font.ITALIC,  
            oldFont.getSize());  
    }  
    public JButton createButtonOk() {  
        JButton b = super.createButtonOk();  
        b.setIcon(getIcon("images/cherry-large.gif"));  
        return b;  
    }  
    public JButton createButtonCancel() {  
        JButton b = super.createButtonCancel();  
        b.setIcon(getIcon("images/cherry-large-down.gif"));  
        return b;  
    }  
}
```

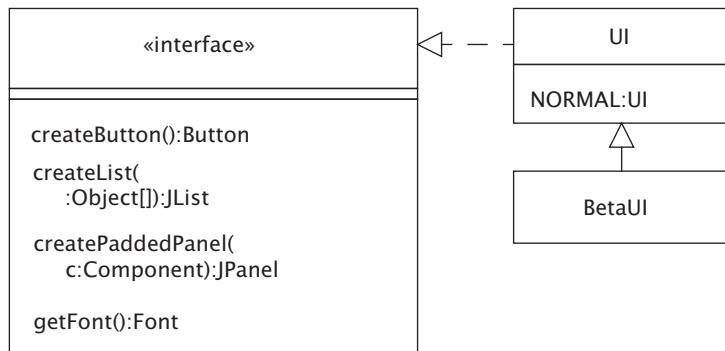
Ce code adopte comme approche d'utiliser autant que possible les méthodes de la classe de base.

Solution 17.2

Une solution apportant une conception plus robuste serait de spécifier les méthodes de création attendues et les propriétés standard de GUI dans une interface, comme illustré Figure B.20.

Figure B.20

Cette conception de classes de factory abstraites pour les composants de GUI réduit la dépendance des sous-classes vis-à-vis des modificateurs des méthodes de la classe UI.

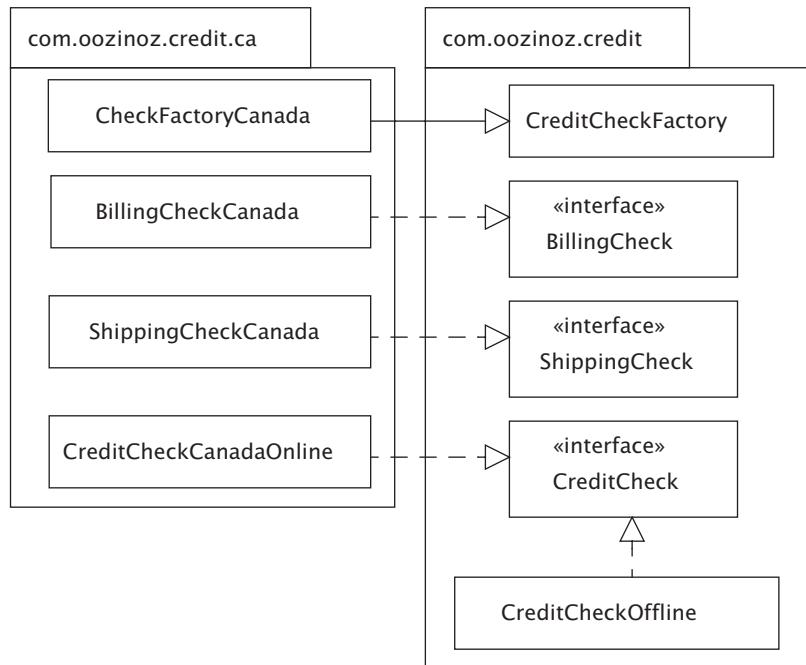


Solution 17.3

La Figure B.21 présente une solution possible pour fournir dans Credit.Canada des classes concrètes qui implémentent les interfaces et les classes abstraites de Credit.

Figure B.21

Le package com.oozinoz.credit.ca fournit une famille de classes concrètes qui opèrent une variété de vérifications lors d'un appel provenant du Canada.



Une subtilité est que vous n'avez besoin que d'une classe concrète pour une vérification de crédit hors ligne car, chez Oozinoz, ce contrôle est le même quelle que soit la provenance des appels.

Solution 17.4

Voici une solution possible :

```
package com.oozinoz.credit.ca;
import com.oozinoz.check.*;
public class CheckFactoryCanada extends CheckFactory {
    public BillingCheck createBillingCheck() {
        return new BillingCheckCanada();
    }
    public CreditCheck createCreditCheck() {
        if (isAgencyUp())
            return new CreditCheckCanadaOnline();
        return new CreditCheckOffline();
    }
    public ShippingCheck createShippingCheck() {
        return new ShippingCheckCanada();
    }
}
```

Votre solution devrait :

- implémenter les méthodes `create...` pour les méthodes héritées de la classe abstraite `CreditCheckFactory` ;
- avoir l'interface appropriée pour le type de retour de chaque méthode `create...` ;
- retourner un objet `CreditCheckOffline` si l'organisme de crédit n'est pas disponible.

Solution 17.5

Voici une justification possible. Le placement de classes spécifiques à un pays dans des packages distincts permet aux développeurs de la société Oozinoz d'organiser les applications et le travail de développement. De plus, les packages propres à chaque pays sont indépendants. Nous pouvons être sûrs que, par exemple, les classes propres aux Etats-Unis n'ont aucun impact sur les classes spécifiques au Canada. Nous pouvons aussi facilement ajouter des fonctionnalités pour de nouveaux pays. Par exemple, si nous commençons à travailler avec Mexico, nous pouvons créer un nouveau package qui fournira les services de contrôle dont nous avons besoin d'une manière cohérente pour ce pays.

Cela apporte l'avantage supplémentaire de nous permettre d'assigner le package `credit.mx` à un développeur possédant une expérience de travail avec des services et des données du Mexique.

Voici un argument contre. Bien que cette séparation soit valable en théorie, elle est excessive dans la pratique. Je préférerais avoir un package avec toutes les classes, du moins jusqu'à ce nous ayons neuf ou dix pays à gérer. La répartition de ces classes dans plusieurs packages multiplie par trois, si ce n'est plus, le travail de gestion de la configuration lorsque je dois implémenter un changement qui couvre tous ces packages.

PROTOTYPE

Solution 18.1

Voici quelques avantages de cette conception :

- Nous pouvons créer de nouveaux objets factory sans avoir à créer une nouvelle classe. Nous pourrions même créer un nouveau kit de GUI lors de l'exécution.
- Nous pouvons produire un nouveau factory par copie en apportant de légères modifications. Par exemple, nous pouvons faire en sorte que le kit de GUI d'une version soit identique au kit normal, excepté pour la police. L'approche avec **PROTOTYPE** permet aux boutons et autres composants d'un nouveau factory d'hériter de valeurs, telles que des couleurs, d'un précédent factory.

Voici quelques inconvénients :

- L'approche avec **PROTOTYPE** permet de changer des valeurs, par exemple pour les couleurs et les polices, pour chaque factory, mais elle ne permet pas de produire de nouveaux kits ayant d'autres comportements.
- La raison de stopper la prolifération des classes de kit de GUI n'est pas claire. En quoi est-elle un problème ? Nous devons placer le code d'initialisation du kit quelque part, probablement dans des méthodes statiques de la classe `UIKit` proposée. Cette approche ne diminue pas réellement la quantité de code à gérer.

Quelle est la réponse correcte ? Dans une telle situation, il peut être utile d'expérimenter. Ecrivez du code qui suit les deux conceptions et évaluez leur comportement dans la pratique. Il y aura des situations où les membres d'une équipe seront en désaccord quant à la direction à suivre. C'est une bonne chose : c'est le signe que vous mettez le doigt sur certains problèmes et que vous discutez conception.

Si vous n'êtes jamais en désaccord, il est probable que vous ne soyiez pas en train d'élaborer la meilleure conception — pour les cas où vous n'êtes pas d'accord, même après une analyse mûrement réfléchie, vous pouvez recourir à un architecte, un chef concepteur ou une tierce partie neutre pour prendre une décision.

Solution 18.2

Une façon de résumer la fonction de `clone()` est "nouvel objet, mêmes champs". La méthode `clone()` crée un nouvel objet en utilisant la même classe et les mêmes types d'attributs que pour l'objet original. Le nouvel objet reçoit aussi les mêmes valeurs de champs que l'original. Si ces champs sont des types de base, tels que des entiers, les valeurs sont copiées. Si toutefois ce sont des références, ce sont les *références* qui sont copiées.

L'objet que la méthode `clone()` crée est une copie partielle (*shallow copy*), elle partage avec l'original tout objet subordonné. Une copie complète (*deep copy*) inclurait des copies entières de tous les attributs de l'objet parent. Par exemple, si vous clonez un objet A qui pointe vers un objet B, une copie partielle créera un nouveau A' qui pointe vers l'objet B original ; une copie complète créera un nouvel A' pointant vers un nouveau B'. Si vous voulez une copie complète, vous devrez implémenter votre propre méthode qui effectuera ce que vous souhaitez.

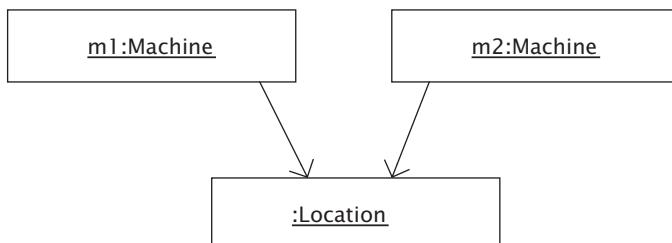
Notez que, pour utiliser `clone()`, vous devez déclarer votre classe comme implémentant `Cloneable`. Cette *interface de marquage* ne possède aucune méthode mais sert d'indicateur pour signaler que vous supportez `clone()` de manière intentionnelle.

Solution 18.3

Le code suggéré ne gardera que trois objets, comme le montre la Figure B.22.

Figure B.22

Une conception insuffisante pour le clonage peut créer une copie incomplète qui partage certains objets avec son prototype.



La version actuelle de la méthode `clone()` pour `MachineSimulator` appelle `super.clone()`, que la classe `Object` implémente. Cette méthode crée un nouvel objet avec les mêmes champs. Des types primitifs, tels que les champs d'instance `int` dans `MachineSimulator`, sont copiés. De plus, les références d'objets, telles que le champ `Location` dans `MachineSimulator`, sont copiées. Notez que c'est la *référence* qui est copiée, pas l'objet. Cela signifie que `Object.clone()` crée la situation illustrée Figure B.22.

Supposez que vous changez la travée et les coordonnées d'emplacement de la deuxième machine. Etant donné qu'il n'y a qu'un objet `Location`, cette modification change l'emplacement des deux machines simulées !

Solution 18.4

Voici une solution possible :

```
public OzPanel copy2() {  
    OzPanel result = new OzPanel();  
    result.setBackground(this.getBackground());  
    result.setForeground(this.getForeground());  
    result.setFont(this.getFont());  
    return result;  
}
```

Les deux méthodes `copy()` et `copy2()` exonèrent les clients de `OzPanel` d'avoir à invoquer un constructeur et à supporter le concept de `PROTOTYPE`. Toutefois, l'approche manuelle de `copy2()` offre peut-être une meilleure sécurité. Elle s'appuie sur le fait de savoir quels sont les attributs importants à copier, mais évite d'avoir à copier des attributs dont vous ignorez tout.

MEMENTO

Solution 19.1

Voici une implémentation possible de `undo()` pour `FactoryModel` :

```
public boolean canUndo() {  
    return mementos.size() > 1;  
}  
public void undo() {  
    if (!canUndo()) return;  
    mementos.pop();  
    notifyListeners();  
}
```

Ce code veille à ignorer les requêtes `undo()` si la pile se retrouve dans son état initial avec un seul memento. Le sommet de la pile est toujours l'état courant, aussi le code `undo()` doit-il seulement effectuer un `pop()` pour exposer le memento précédent.

Lorsque vous écrivez une méthode `createMemento()`, vous devriez faire en sorte qu'elle retourne toutes les informations nécessaires pour reconstruire l'objet récepteur. Dans cet exemple, un simulateur de machine peut se reconstruire à partir d'un clone, et un simulateur d'usine peut se reconstruire à partir d'une liste de clones de simulateurs de machines.

Solution 19.2

Une solution possible est :

```
public void stateChanged(ChangeEvent e) {  
    machinePanel().removeAll();  
    List locations = factoryModel.getLocations();  
    for (int i = 0; i < locations.size(); i++) {  
        Point p = (Point) locations.get(i);  
        machinePanel().add(createPictureBox(p));  
    }  
    undoButton().setEnabled(factoryModel.canUndo());  
    repaint();  
}
```

Chaque fois que l'état change, ce code reconstruit entièrement la liste de machines dans `machinePanel()`.

Solution 19.3

Stocker un memento en tant qu'objet suppose que l'application soit toujours en train de s'exécuter lorsque l'utilisateur souhaite restaurer l'objet original. Voici les raisons qui pourraient vous inciter à stocker un memento sous forme persistante :

- pouvoir restaurer l'état d'un objet si le système plante ;
- permettre à l'utilisateur de quitter le système et de poursuivre son travail ultérieurement ;
- pouvoir reconstruire un objet sur un autre ordinateur.

Solution 19.4

Une solution possible est :

```
public void restore(Component source) throws Exception {
    JFileChooser dialog = new JFileChooser();
    dialog.showOpenDialog(source);
    if (dialog.getSelectedFile() == null)
        return;
    FileInputStream out = null;
    ObjectInputStream s = null;
    try {
        out = new FileInputStream(dialog.getSelectedFile());
        s = new ObjectInputStream(out);
        ArrayList list = (ArrayList) s.readObject();
        factoryModel.setLocations(list);
    } finally {
        if (s != null)
            s.close();
    }
}
```

Ce code est presque identique à la méthode `save()`, sauf que la méthode `restore()` doit demander au modèle d'usine de lui transmettre par poussé (*push*) la liste de machines récupérée.

Solution 19.5

Encapsuler limite l'accès à l'état et aux opérations d'un objet. Le fait d'enregistrer un objet, tel qu'une collection d'emplacements, sous forme textuelle expose les données de l'objet et permet à n'importe qui disposant d'une éditeur de texte de changer l'état de l'objet. Par conséquent, enregistrer un objet au format XML viole la règle d'encapsulation, dans une certaine mesure en tout cas.

Selon votre application, une violation de la règle d'encapsulation par un stockage persistant peut poser un problème dans la pratique. Pour y remédier, vous pourriez limiter l'accès aux données, ce qui est courant dans une base de données relationnelle. Vous pourriez sinon chiffrer les données, ce qui est courant lors de la transmission de texte HTML sensible. L'important ici n'est pas de savoir si les concepts d'encapsulation et de memento s'appliquent à une conception mais plutôt de garantir l'intégrité des données tout en supportant le stockage et la transmission de données.

Introduction aux opérations

Solution 20.1

Le pattern CHAIN OF RESPONSIBILITY distribue une opération à travers une chaîne d'objets. Chaque méthode implémente le service de l'opération directement ou transmet les appels à l'objet suivant dans la chaîne.

Solution 20.2

Voici une liste complète des modificateurs de méthodes Java avec une définition informelle de chacun :

- **public** : accès permis à tous les clients.
- **protected** : accès permis au sein du package de déclaration et aux sous-classes de la classe.
- **private** : accès permis uniquement au sein de la classe.
- **abstract** : pas d'implémentation fournie.
- **static** : associée à la classe dans son ensemble, pas à des objets individuels.
- **final** : ne peut être remplacée.
- **synchronized** : la méthode acquiert un accès au moniteur de l'objet ou de la classe, si elle est statique.
- **native** : implémentée ailleurs dans du code dépendant d'une plate-forme.
- **strictfp** : les expressions **double** et **float** évaluées strictement d'après les règles FP, nécessitant que les résultats intermédiaires soient valides selon les standards IEEE.

Bien que certains développeurs puissent être tentés d'utiliser tous ces modificateurs dans une même définition de méthode, plusieurs règles limitent les possibilités de combinaison.

Solution 20.3

Cela dépend.

Avec les versions antérieures à Java 5, si vous changez la valeur de retour de `Bitmap.clone()`, le code ne sera pas compilé. La signature de `clone()` correspond à la signature de `Object.clone()`, et les types de retour doivent donc également correspondre.

Depuis Java 5, la définition du langage a changé pour supporter des **types de retour covariants**, permettant à une sous-classe de déclarer un type de retour plus spécifique.

Solution 20.4

Voici un argument *pour* le fait d'omettre les déclarations d'exceptions dans les en-têtes de méthodes. Tout d'abord, il convient de noter que Java n'impose pas que les méthodes déclarent toutes les exceptions qui pourraient être générées. N'importe quelle méthode pourrait, par exemple, rencontrer un pointeur `null` et générer une exception non déclarée. Il ne serait donc pas pratique d'obliger les programmeurs à déclarer toutes les exceptions possibles. Les applications requièrent une stratégie pour pouvoir gérer toutes les exceptions, laquelle ne peut être remplacée en demandant aux développeurs de déclarer certains types d'exceptions.

Un argument *contre* est que les programmeurs ont besoin d'aide. Il est vrai que l'architecture d'une application doit disposer d'une stratégie efficace de gestion des exceptions. De toute évidence, il ne serait pas commode de forcer les développeurs à déclarer dans chaque méthode l'éventualité d'un problème courant, tel que des pointeurs `null`. Mais, pour certaines erreurs, telles qu'un problème d'ouverture de fichier, demander à l'invocateur d'une méthode de gérer les exceptions possibles pourrait être utile. C# résout ce dilemme en éliminant toute déclaration d'exception de l'en-tête des méthodes.

Solution 20.5

La figure illustre un algorithme — la procédure déterminant si un modèle objet est un arbre —, deux opérations — apparaissant en tant que signatures dans la classe `MachineComponent` —, et quatre méthodes.

TEMPLATE METHOD

Solution 21.1

Votre programme complété devrait ressembler à ce qui suit :

```
package app.templateMethod;
import java.util.Comparator;
import com.oozinoz.firework.Rocket;
public class ApogeeComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        Rocket r1 = (Rocket) o1;
        Rocket r2 = (Rocket) o2;
        return Double.compare(r1.getApogee(), r2.getApogee());
    }
}
```

et

```
package app.templateMethod;
import java.util.Comparator;
import com.oozinoz.firework.Rocket;
public class NameComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        Rocket r1 = (Rocket) o1;
        Rocket r2 = (Rocket) o2;
        return r1.toString().compareTo(r2.toString());
    }
}
```

Solution 21.2

Le code de `markMoldIncomplete()` passe des informations relatives au moule incomplet au gestionnaire du matériel. Voici une solution possible :

```
package com.oozinoz.ozAster;
import aster.*;
import com.oozinoz.businessCore.*;
public class OzAsterStarPress extends AsterStarPress {
    public MaterialManager getManager() {
        return MaterialManager.getManager();
    }
    public void markMoldIncomplete(int id) {
        getManager().setMoldIncomplete(id);
    }
}
```

Solution 21.3

Vous avez besoin ici d'un hook. Vous pourriez formuler votre demande comme ceci : "Vous serait-il possible d'insérer un appel dans votre méthode `shutDown()`, après le déchargeement de la pâte et avant le rinçage de la machine ? Si vous le nommiez quelque chose comme `collectPaste()`, je pourrais l'utiliser pour récupérer la pâte que nous réutilisons chez Oozinoz."

Les développeurs de chez Aster discuteraient probablement avec vous du nom à donner à la méthode. L'intérêt de demander un hook dans un TEMPLATE METHOD est que cela permet à votre code d'être beaucoup plus robuste que si vous deviez vous accommoder d'un code existant inadéquat.

Solution 21.4

La méthode `getPlanner()` de la classe `Machine` devrait tirer parti de la méthode abstraite `createPlanner()` :

```
public MachinePlanner getPlanner() {  
    if (planner == null)  
        planner = createPlanner();  
    return planner;  
}
```

Ce code implique que vous ajoutiez un champ `planner` à la classe `Machine`. Après avoir ajouté cet attribut et la méthode `getPlanner()`, vous pouvez les supprimer dans les sous-classes.

Cette refactorisation crée un TEMPLATE METHOD. La méthode `getPlanner()` procède à une initialisation paresseuse de la variable `planner`, s'appuyant uniquement sur l'étape `createPlanner()` fournie par les sous-classes.

STATE

Solution 22.1

Comme le montre la machine à états, lorsque la porte est ouverte, le fait de toucher le bouton la place dans l'état `StayOpen`, et le fait de toucher le bouton une seconde fois la fait se fermer.

Solution 22.2

Votre code devrait ressembler à ce qui suit :

```
public void complete() {  
    if (state == OPENING)
```

```

        setState(OPEN);
    else if (state == CLOSING)
        setState(CLOSED);
}
public void timeout() {
    setState(CLOSING);
}

```

Solution 22.3

Votre code devrait ressembler à ce qui suit :

```

package com.oozinoz.carousel;
public class DoorClosing extends DoorState {
    public DoorClosing(Door2 door) {
        super(door);
    }
    public void touch() {
        door.setState(door.OPENING);
    }
    public void complete() {
        door.setState(door.CLOSED);
    }
}

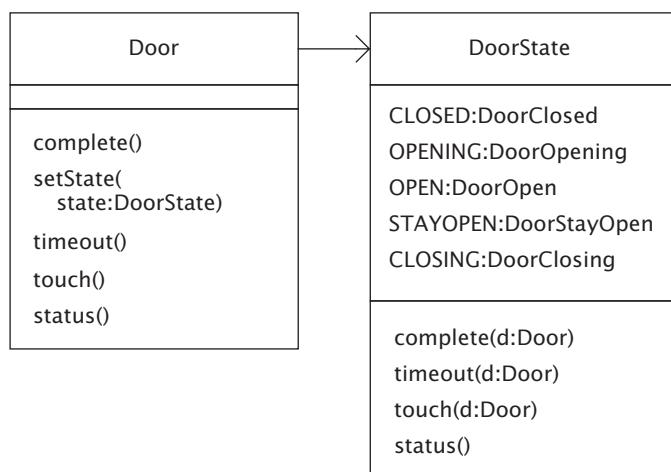
```

Solution 22.4

La Figure B.23 illustre le diagramme complété.

Figure B.23

Cette conception rend les objets DoorState constants. Les méthodes de transition d'état de DoorState actualisent l'état d'un objet Door qu'elles reçoivent comme paramètre.



STRATEGY

Solution 23.1

La Figure B.24 présente une solution.

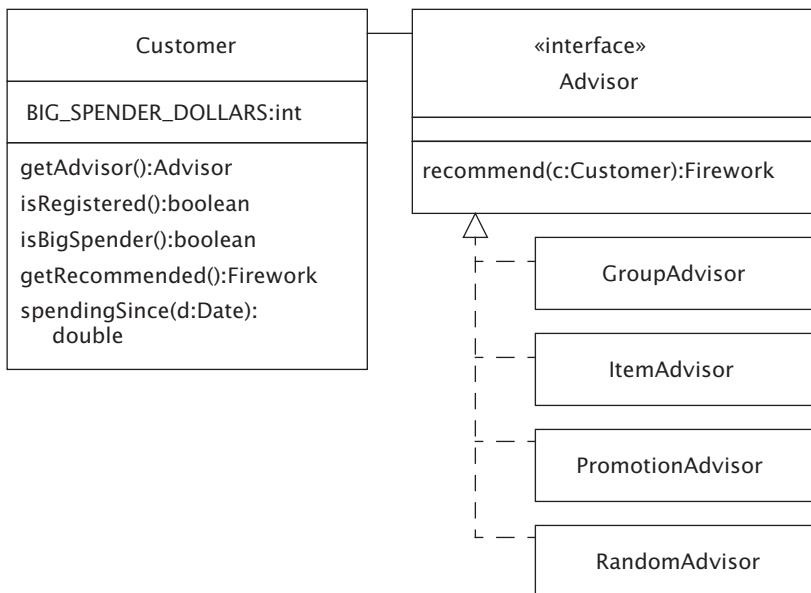


Figure B.24

La politique publicitaire d'Oozinoz inclut quatre stratégies qui apparaissent sous la forme de quatre implémentations de l'interface Advisor.

Solution 23.2

Les classes `GroupAdvisor` et `ItemAdvisor` sont des instances de ADAPTER, fournissant l'interface qu'un client attend en utilisant les services d'une classe dont l'interface est différente.

Solution 23.3

Votre code pourrait ressembler à ceci :

```

public Firework getRecommended() {
    return getAdvisor().recommend(this);
}
  
```

Une fois que l'attribut `advisor` est connu, le polymorphisme fait le reste.

Solution 23.4

Une routine de tri réutilisable est-elle un exemple de TEMPLATE METHOD ou de STRATEGY ?

D'après *Design Patterns*, TEMPLATE METHOD laisse les sous-classes redéfinir certaines étapes d'un algorithme. Mais la méthode `Collections.sort()` ne s'appuie pas sur les sous-classes. Elle utilise une instance de `Comparator`. Chaque instance de `Comparator` fournit une nouvelle méthode et donc un nouvel algorithme et une nouvelle stratégie. Cette méthode est donc un bon exemple de STRATEGY.

Il existe de nombreux algorithmes de tri, mais `Collections.sort()` en utilise un seul, le tri rapide (*quick sort*). Changer d'algorithme signifierait utiliser à la place le tri par tas (*heap sort*) ou le tri à bulles (*bubble sort*). L'objectif de STRATEGY est de vous permettre d'utiliser différents algorithmes, ce qui ne se produit pas ici. L'objectif de TEMPLATE METHOD est de vous permettre d'insérer une étape dans un algorithme, ce qui correspond précisément au fonctionnement de la méthode `sort()`.

COMMAND

Solution 24.1

Nombre d'applications Java Swing appliquent le pattern MEDIATOR enregistrant un seul objet pour recevoir tous les événements GUI. Cet objet sert de médiateur aux composants qui interagissent et traduit l'entrée utilisateur en commandes d'objets du domaine.

Solution 24.2

Voici à quoi pourrait ressembler votre code :

```
package com.oozinoz.visualization;
import java.awt.event.*;
import javax.swing.*;
import com.oozinoz.ui.*;
public class Visualization2 extends Visualization {
    public static void main(String[] args) {
        Visualization2 panel = new Visualization2(UI.NORMAL);
        JFrame frame = SwingFacade.launch(
            panel,
            "Operational Model");
        frame.setJMenuBar(panel.menus());
        frame.setVisible(true);
    }
}
```

```
public Visualization2(UI ui) {
    super(ui);
}
public JMenuBar menus() {
    JMenuBar menuBar = new JMenuBar();
    JMenu menu = new JMenu("File");
    menuBar.add(menu);
    JMenuItem menuItem = new JMenuItem("Save As...");
    menuItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            save();
        }
    });
    menu.add(menuItem);
    menuItem = new JMenuItem("Restore From...");
    menuItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            restore();
        }
    });
    menu.add(menuItem);
    return menuBar;
}
public void save() {
    try {
        mediator.save(this);
    } catch (Exception ex) {
        System.out.println("Echec de sauvegarde : " +
                           ex.getMessage());
    }
}
public void restore() {
    try {
        mediator.restore(this);
    } catch (Exception ex) {
        System.out.println("Echec de restauration : " +
                           ex.getMessage());
    }
}
```

Bien que la méthode `actionPerformed()` nécessite un argument `ActionEvent`, vous pouvez l'ignorer sans problème. La méthode `menus()` enregistre une seule instance d'une classe anonyme avec l'option de menu Save et une seule instance d'une autre classe anonyme avec l'option de menu Load. Lorsque ces méthodes sont appelées, il n'y a aucun doute possible quant à la source de l'événement.

Solution 24.3

La méthode `testSleep()` passe la commande `doze` à la méthode `time()` :

```
package app.command;
import com.oozinoz.robotInterpreter.Command;
import com.oozinoz.utility.CommandTimer;
import junit.framework.TestCase;
public class TestCommandTimer extends TestCase {
    public void testSleep() {
        Command doze = new Command() {
            public void execute() {
                try {
                    Thread.sleep(
                        2000 + Math.round(10 * Math.random()));
                } catch (InterruptedException ignored) {
                }
            }
        };
        long actual = CommandTimer.time(doze);
        long expected = 2000;
        long delta = 5;
        assertTrue(
            "Devrait être " + expected + " +/- " + delta + " ms",
            expected - delta <= actual
            && actual <= expected + delta);
    }
}
```

Solution 24.4

Votre code pourrait ressembler à ceci :

```
public void shutDown() {
    if (inProcess()) {
        stopProcessing();
        moldIncompleteHook.execute(this);
    }
    usherInputMolds();
    dischargePaste();
    flush();
}
```

Notez que ce code ne s'occupe pas de vérifier si `moldIncompleteHook` est `null`, puisque cet argument est toujours défini avec un véritable objet `Hook` (initialement, il est défini avec un objet `NullHook` qui ne fait rien, mais un utilisateur peut installer un hook différent).

Vous pourriez l'utiliser comme suit :

```
package app.templateMethod;
import com.oozinoz.businessCore.*;
import aster2.*;
public class ShowHook {
    public static void main(String[] args) {
        AsterStarPress p = new AsterStarPress();
        Hook h = new Hook() {
            public void execute(AsterStarPress p) {
                MaterialManager m = MaterialManager.getManager();
                m.setMoldIncomplete(p.getCurrentMoldID());
            }
        };
        p.setMoldIncompleteHook(h);
    }
}
```

Solution 24.5

Dans FACTORY METHOD, un client sait *quand* créer un nouvel objet mais ne sait pas *quel type* d'objet. Ce pattern place la création d'un objet dans une méthode permettant au client de ne pas connaître la classe à instancier. Ce principe est également présent dans ABSTRACT FACTORY.

Solution 24.6

L'objectif du pattern MEMENTO est d'assurer le stockage et la restauration de l'état d'un objet. Typiquement, vous pouvez ajouter un nouveau memento à une pile pour chaque exécution d'une commande, puis les récupérer et les appliquer chaque fois qu'un utilisateur a besoin de défaire des commandes.

INTERPRETER

Solution 25.1

La méthode `execute()` de la classe `ForCommand` devrait ressembler au code suivant :

```
private void execute(MachineComponent mc) {
    if (mc instanceof Machine) {
        Machine m = (Machine) mc;
        variable.assign(new Constant(m));
        body.execute();
        return;
    }
}
```

```

MachineComposite comp = (MachineComposite) mc;
List children = comp.getComponents();
for (int i = 0; i < children.size(); i++) {
    MachineComponent child =
        (MachineComponent) children.get(i);
    execute(child);
}
}

```

Le code de `execute()` parcourt un composite de machines. Lorsqu'il rencontre un nœud feuille — une machine —, il assigne la variable à la machine et exécute la commande `body` de l'objet `ForMachine`.

Solution 25.2

Une solution possible est :

```

public void execute() {
    if (term.eval() != null)
        body.execute();
    else
        elseBody.execute();
}

```

Solution 25.3

Voici une manière d'écrire la classe `WhileCommand.java` :

```

package com.oozinoz.robotInterpreter2;
public class WhileCommand extends Command {
    protected Term term;
    protected Command body;
    public WhileCommand(Term term, Command body) {
        this.term = term;
        this.body = body;
    }
    public void execute() {
        while (term.eval() != null)
            body.execute();
    }
}

```

Solution 25.4

Voici un exemple de réponse. L'objectif du pattern INTERPRETER est de vous permettre de composer des objets exécutables à partir d'une hiérarchie de classes fournissant différentes interprétations d'une opération commune. L'objectif du pattern COMMAND est simplement d'encapsuler une requête dans un objet.

Un objet interpréteur peut-il fonctionner comme une commande ? Bien entendu ! Le choix du pattern à utiliser dépend de votre but. Etes-vous en train de créer un kit d'outils pour composer des objets exécutables ou d'encapsuler une requête dans un objet ?

Introduction aux extensions

Solution 26.1

En mathématique, un cercle est un cas spécial d'ellipse. Mais en programmation OO, une ellipse ne possède pas le même comportement qu'un cercle. Par exemple, sa largeur peut faire le double de sa hauteur, ce qui est impossible pour un cercle. Si ce comportement est important pour votre programme, un objet `Circle` ne fonctionnera pas comme un objet `Ellipse` et constituera donc une violation de LSP.

Notez qu'il peut en aller autrement pour les objets **immuables**. C'est simplement un domaine dans lequel les mathématiques naïves s'accordent mal avec la sémantique des hiérarchies de types standard.

Solution 26.2

L'expression `tub.getLocation().isUp()` peut donner lieu à des erreurs de programmation si certaines subtilités entourent la valeur de la propriété `Location` d'un objet `Tub`. Par exemple, cette propriété pourrait être `null` ou être un objet `Robot` si le bac est en transit. Dans le premier cas, l'évaluation de `tub.getLocation().isUp()` génère une exception ; dans le second, le problème peut même être pire puisque nous tentons d'utiliser un robot pour récupérer un bac à partir de lui-même. Ces problèmes potentiels sont gérables, mais voulons-nous qu'un tel code de gestion soit placé dans la méthode contenant l'expression `tub.getLocation().isUp()` ? Non. Le code nécessaire se trouve peut-être déjà dans la classe `Tub`. Si ce n'est pas le cas, il devrait en tout cas s'y trouver pour nous éviter d'avoir à coder les mêmes subtilités dans d'autres méthodes qui interagissent avec les bacs.

Solution 26.3

Voici un exemple :

```
public static String getZip(String address) {  
    return address.substring(address.length() - 5);  
}
```

Cette méthode comporte quelques indicateurs, dont un indicateur *obsession primitive* (utilisation d'une chaîne pour contenir plusieurs attributs).

Solution 26.4

Voici un exemple de solution :

<i>Exemple</i>	<i>Pattern à l'œuvre</i>
Le concepteur d'une simulation pyrotechnique établit une interface qui définit le comportement que doit présenter votre objet pour pouvoir prendre part à la simulation	ADAPTER
Un kit d'outils permet de composer des objets exécutables lors de l'exécution	INTERPRETER
Une super-classe possède une méthode qui dépend de sous-classes pour fournir une étape manquante	TEMPLATE METHOD
Un objet vous permet d'étendre son comportement en acceptant une méthode encapsulée dans un autre objet et en invoquant cette méthode au moment approprié	COMMAND
Un générateur de code insère un comportement qui donne l'illusion qu'un objet s'exécutant sur une autre machine est local	PROXY
Une conception vous permet d'enregistrer des listeners pour des rappels qui auront lieu lorsqu'un objet change	OBSERVER
Une conception vous permet de définir des opérations abstraites qui dépendent d'une interface spécifique et d'ajouter de nouveaux drivers qui satisfont aux exigences de cette interface	BRIDGE

DECORATOR

Solution 27.1

Voici une solution possible :

```
package com.oozinoz.filter;
import java.io.*;
public class RandomCaseFilter extends OozinozFilter {
    public RandomCaseFilter(Writer out) {
        super(out);
    }
    public void write(int c) throws IOException {
```

```
        out.write(Math.random() < .5
            ? Character.toLowerCase((char) c)
            : Character.toUpperCase((char) c));
    }
}
```

Une casse aléatoire accroche l'attention. Considérez le programme suivant :

```
package app.decorator;
import java.io.BufferedReader;
import java.io.IOException;
import com.oozinoz.filter.ConsoleWriter;
import com.oozinoz.filter.RandomCaseFilter;
public class ShowRandom {
    public static void main(String[] args)
        throws IOException {
        BufferedReader w =
            new BufferedReader(
                new RandomCaseFilter(new ConsoleWriter()));
        w.write("buy two packs now and get a "
            + "zippie pocket rocket -- free!");
        w.newLine();
        w.close();
    }
}
```

Ce programme produit quelque chose comme ceci :

```
bUy tWO pAcks NOw ANd geT A ZiPPIE PoCkEt RocKeT -- frEe!
```

Solution 27.2

Voici une solution possible :

```
package com.oozinoz.filter;
import java.io.Writer;
public class ConsoleWriter extends Writer {
    public void close() {}
    public void flush() {}
    public void write(
        char[] buffer, int offset, int length) {
        for (int i = 0; i < length; i++)
            System.out.print(buffer[offset + i]);
    }
}
```

Solution 27.3

Voici une solution possible :

```
package com.oozinoz.function;
public class Exp extends Function {
    public Exp(Function f) {
        super(f);
    }
    public double f(double t) {
        return Math.exp(sources[0].f(t));
    }
}
```

Solution 27.4

Voici une solution possible :

```
package app.decorator.brightness;
import com.oozinoz.function.Function;
public class Brightness extends Function {
    public Brightness(Function f) {
        super(f);
    }
    public double f(double t) {
        return Math.exp(-4 * sources[0].f(t))
            * Math.sin(Math.PI * sources[0].f(t));
    }
}
```

ITERATOR

Solution 28.1

La routine `display()` lance un nouveau thread qui peut se réveiller à n'importe quel moment, mais l'appel `sleep()` veille à ce que `run()` s'exécute pendant que `display()` est inactive. Le résultat indique que lorsqu'elle s'exécute, la méthode `display()` garde le contrôle pendant une itération, affichant la liste à partir de l'indice 0 :

Mixer1201

Ensuite, le second thread devient actif et place Fuser1101 au début de la liste, décalant toutes les autres machines d'un indice. En particulier, Mixer1201 passe de l'indice 0 à l'indice 1.

Lorsque le thread principal reprend le contrôle, `display()` affiche le reste de la liste depuis l'indice 1 jusqu'à la fin :

```
Mixer1201  
ShellAssembler1301  
StarPress1401  
UnloadBuffer1501
```

Solution 28.2

Un argument *contre l'utilisation des méthodes synchronized()* est qu'elles peuvent donner lieu à des résultats erronés — si l'itération utilise une boucle `for` — ou faire planter le programme, à moins de prévoir une logique qui intercepte l'exception `InvalidOperationException` générée.

Un argument *contre l'approche avec verrouillage* est que les conceptions qui assurent une itération avec sécurité inter-thread s'appuient sur la coopération entre les threads pouvant accéder à la collection. Les méthodes `synchronized()` servent justement dans le cas où les threads ne coopèrent pas.

Ni les méthodes `synchronized()` ni le support du verrouillage intégrés à Java ne peuvent rendre le développement multithread aisé et infaillible.

Solution 28.3

Comme décrit au Chapitre 16, les itérateurs constituent un exemple classique du pattern FACTORY METHOD. Un client qui souhaite un énumérateur pour une instance de `ProcessComponent` sait quand créer l'itérateur, et la classe réceptrice sait quelle classe instancier.

Solution 28.4

Voici une solution possible :

```
public Object next() {  
    if (peek != null) {  
        Object result = peek;  
        peek = null;  
        return result;  
    }  
    if (!visited.contains(head)) {  
        visited.add(head);  
        if (shouldShowInterior()) return head;  
    }  
    return nextDescendant();  
}
```

VISITOR

Solution 29.1

La différence se situe au niveau du type de l'objet `this`. La méthode `accept()` invoque la méthode `visit()` d'un objet `MachineVisitor`. La méthode `accept()` de la classe `Machine` recherche une méthode `visit()` possédant la signature `visit(:Machine)`, tandis que la méthode `accept()` de la classe `MachineComposite` recherche une méthode `visit()` possédant la signature `visit(:MachineComposite)`.

Solution 29.2

Voici une solution possible :

```
package app.visitor;
import com.oozinoz.machine.MachineComponent;
import com.oozinoz.machine.OozinozFactory;
public class ShowFindVisitor {
    public static void main(String[] args) {
        MachineComponent factory = OozinozFactory.dublin();
        MachineComponent machine = new FindVisitor().find(
            factory, 3404);
        System.out.println(machine != null ?
            machine.toString() : "Introuvable");
    }
}
```

Solution 29.3

Voici une solution possible :

```
package app.visitor;
import com.oozinoz.machine.*;
import java.util.*;
public class RakeVisitor implements MachineVisitor {
    private Set leaves;
    public Set getLeaves(MachineComponent mc) {
        leaves = new HashSet();
        mc.accept(this);
        return leaves;
    }
    public void visit(Machine m) {
        leaves.add(m);
    }
    public void visit(MachineComposite mc) {
```

```
Iterator iter = mc.getComponents().iterator();
while (iter.hasNext())
    ((MachineComponent) iter.next()).accept(this);
}
```

Solution 29.4

Une solution est d'ajouter un argument `Set` à toutes les méthodes `accept()` et `visit()` pour passer l'ensemble des nœuds visités. La classe `ProcessComponent` devrait alors inclure une méthode `accept()` concrète appelant sa méthode `accept()` abstraite, lui passant un nouvel objet `Set` :

```
public void accept(ProcessVisitor v) {
    accept(v, new HashSet());
}
```

La méthode `accept()` des sous-classes `ProcessAlternation`, `ProcessSequence` et `ProcessStep` serait :

```
public void accept(ProcessVisitor v, Set visited) {
    v.visit(this, visited);
}
```

Les développeurs doivent aussi créer des classes avec des méthodes `visit()` qui acceptent l'ensemble `visited`. Il leur revient également de remplir l'ensemble.

Solution 29.5

Voici les alternatives à l'application du pattern VISITOR :

- Ajoutez le comportement dont vous avez besoin à la hiérarchie originale. Vous pouvez le faire si vous communiquez facilement avec ses développeurs ou si vous suivez le principe de propriété collective du code.
- Vous pouvez laisser une classe simplement traverser une structure de machines ou de processus sur laquelle elle doit opérer. Si vous avez besoin de connaître le type d'un enfant du composite, par exemple, vous pouvez utiliser l'opérateur `instanceof`, ou vous pourriez sinon introduire des fonctions booléennes, telles que `isLeaf()` et `isComposite()`.
- Si le comportement que vous voulez ajouter est très différent de celui existant, vous pouvez créer une hiérarchie parallèle. Par exemple, la classe `MachinePlanner` du Chapitre 16, consacré à FACTORY METHOD, place un comportement de planification dans une hiérarchie séparée.

C

Code source d'Oozinoz

Le premier avantage de comprendre les patterns, ou modèles, de conception est qu'ils vous aident à améliorer votre code. Vous obtiendrez un code plus concis, plus simple, plus élégant et plus puissant, et sa maintenance en sera aussi facilitée. Pour en percevoir les réels bénéfices, voyez-les en action dans du code exécuté. Vous devez devenir à l'aise dans la construction ou reconstruction d'une base de code avec des patterns. Il peut être utile de commencer avec des exemples fonctionnels, et ce livre inclut de nombreux exemples d'implémentation illustrant leur emploi dans du code Java. La compilation du code source d'Oozinoz et l'examen des exemples fournis tout au long du livre pour étayer les concepts décrits vous aideront à débuter l'implémentation de patterns dans votre propre code.

Obtention et utilisation du code source

Pour obtenir le code source compagnon de ce livre, rendez-vous sur le site www.oozinoz.com, téléchargez le fichier zip contenant le code source, et décompressez-en le contenu. Vous pouvez le placer n'importe où sur votre système de fichiers. Ce code est gratuit, vous pouvez l'utiliser comme bon vous semble avec pour seule condition de ne pas affirmer en être l'auteur. D'un autre côté, les auteurs et l'éditeur de ce livre ne garantissent en aucune façon que ce code sera utile et adéquat pour quelque objectif que ce soit.

Construction du code d'Oozinoz

Si vous ne disposez pas d'un environnement de développement pour écrire des programmes Java, vous devez en acquérir un et vous familiariser avec pour pouvoir développer, compiler et exécuter vos propres programmes. Vous pouvez acheter un outil de développement, tel qu'Idea d'IntelliJ, ou recourir à des outils open source, tels qu'Eclipse.

Test du code avec JUnit

Les bibliothèques Oozinoz incluent un package `com.oozinoz.testing` conçu pour être utilisé avec JUnit, un environnement de test automatisé. Vous pouvez télécharger JUnit à partir de <http://junit.org>. Si l'emploi de cet environnement ne vous est pas familier, la meilleure façon d'apprendre à vous en servir est de demander de l'aide auprès d'un ami ou d'un collègue. Autrement, vous pouvez vous aider de la documentation en ligne ou trouver un livre sur le sujet. Le temps d'apprentissage est moins rapide qu'avec Ant, par exemple, mais l'étudier vous apportera des compétences qui vous seront profitables pour de nombreuses années.

Localiser les fichiers

Il peut être difficile de trouver le fichier particulier correspondant à un extrait de code présenté dans le livre. Le moyen le plus simple de localiser une application donnée est souvent de rechercher son nom dans l'arborescence du répertoire `oozinoz`. L'organisation de cette arborescence devrait vous permettre de facilement localiser les fichiers voulus. Le Tableau C.1 reprend les sous-répertoires de `oozinoz` avec une description de leur contenu.

Tableau C.1 : Sous-répertoires du répertoire `oozinoz`

<i>Répertoire</i>	<i>Contenu</i>
<code>app</code>	Sous-répertoires des "applications" : fichiers Java sous-jacents aux programmes exécutables. Ils sont généralement organisés par chapitres. Ainsi <code>app.decorator</code> contient du code se rapportant au Chapitre 27, DECORATOR.
<code>aster</code>	Le code source d'Aster, une société fictive.
<code>com</code>	Le code source des applications Oozinoz.
<code>images</code>	Des images utilisées par diverses applications Oozinoz.

Résumé

Les efforts investis dans l'apprentissage des patterns de conception commenceront à porter leurs fruits lorsque vous changerez votre façon d'écrire et de *refactoriser*, ou *restructurer*, du code. Vous pourrez même appliquer directement des patterns dans votre propre code. Réussir à faire fonctionner le code d'Oozinoz sur votre propre machine sera un exercice utile, comme d'apprendre à utiliser des outils open source, tels qu'Ant et JUnit. Apprendre ces derniers et parvenir à faire tourner le code d'Oozinoz, ou de n'importe qui d'autre, peut représenter un certain travail, mais les bénéfices tirés de ces difficultés vous seront profitables pendant de longues années grâce aux nouvelles compétences acquises, lesquelles seront applicables au fur et à mesure de l'apparition de nouvelles techniques et technologies.

Bonne chance ! Si vous rencontrez des problèmes ou êtes bloqué, n'hésitez pas à écrire à l'un de nous deux.

Steve Metsker (Steve.Metsker@acm.org)

William Wake (William.Wake@acm.org)

D

Introduction à UML

Cette annexe est une brève introduction au langage de modélisation UML (*Unified Modeling Language*), que ce livre utilise. UML propose une convention de notation permettant d’illustrer la conception de systèmes orientés objet. Ce langage n’est pas excessivement complexe, mais il est facile d’en sous-estimer la richesse fonctionnelle. Pour une introduction rapide à la plupart de ses fonctionnalités, voyez l’ouvrage *UML Distilled* [Fowler et Scott 2003]. Pour une analyse plus approfondie, reportez-vous à l’ouvrage *The Unified Modeling Language User Guide (le Guide de l’utilisateur UML)* [Booch, Rumbaugh, et Jacobson 1999]. L’apprentissage de nomenclatures et notations standard permet de communiquer au niveau conception et d’être plus efficace.

Classes

La Figure D.1 applique certaines des fonctionnalités d’UML pour illustrer des classes.

Voici quelques directives concernant l’élaboration de diagrammes de classes :

- Dessinez une classe en centrant son nom dans un rectangle. La Figure D.1 montre trois classes : `Firework`, `Rocket` et `simulation.RocketSim`.
- Le langage n’exige pas qu’un diagramme doive tout montrer à propos d’un élément illustré, comme l’ensemble des méthodes d’une classe ou le contenu complet d’un package.

- Signalez un package en alignant son nom sur la gauche dans un rectangle adjoint à un plus grand encadré pouvant illustrer des classes ou d'autres types d'éléments. Par exemple, les classes dans la Figure D.1 se trouvent dans le package Fireworks.
- Lorsqu'une classe est représentée en dehors d'un diagramme de package, ajoutez en préfixe l'espace de nom (*namespace*) séparé du nom de la classe par un point. Par exemple, la Figure D.1 montre que la classe RocketSim se trouve dans le package simulation.
- Vous pouvez spécifier les variables d'instance d'une classe dans une subdivision rectangulaire au-dessous du nom de la classe. La classe Firework possède les variables d'instance name, mass et price. Faites suivre un nom de variable d'un double-point et de son type.
- Vous pouvez signifier qu'une variable d'instance ou une méthode est privée en faisant précéder son nom d'un signe moins (-). Un signe (+) indique qu'elle est publique, et un signe dièse (#) signale qu'elle est protégée.
- Vous pouvez spécifier les méthodes d'une classe dans un second rectangle au-dessous du nom de la classe.
- Quand une méthode attend des paramètres en entrée, vous pouvez les indiquer, comme le fait la méthode lookup() dans la Figure D.1
- Dans les signatures de méthode, les variables sont spécifiées par leur nom suivi du signe double-point et de leur type. Vous pouvez omettre ou abréger le nom si le type suggère clairement le rôle de la variable.
- Soulignez le nom d'une méthode pour signifier qu'elle est statique, comme c'est le cas de la méthode lookup() dans la Figure D.1.
- Inscrivez des notes dans un encadré comportant un coin rabattu. Le texte contenu peut comprendre des commentaires, des contraintes ou du code. Reliez la note aux autres éléments du diagramme par un trait en pointillé. Les notes peuvent apparaître dans n'importe quel diagramme UML.

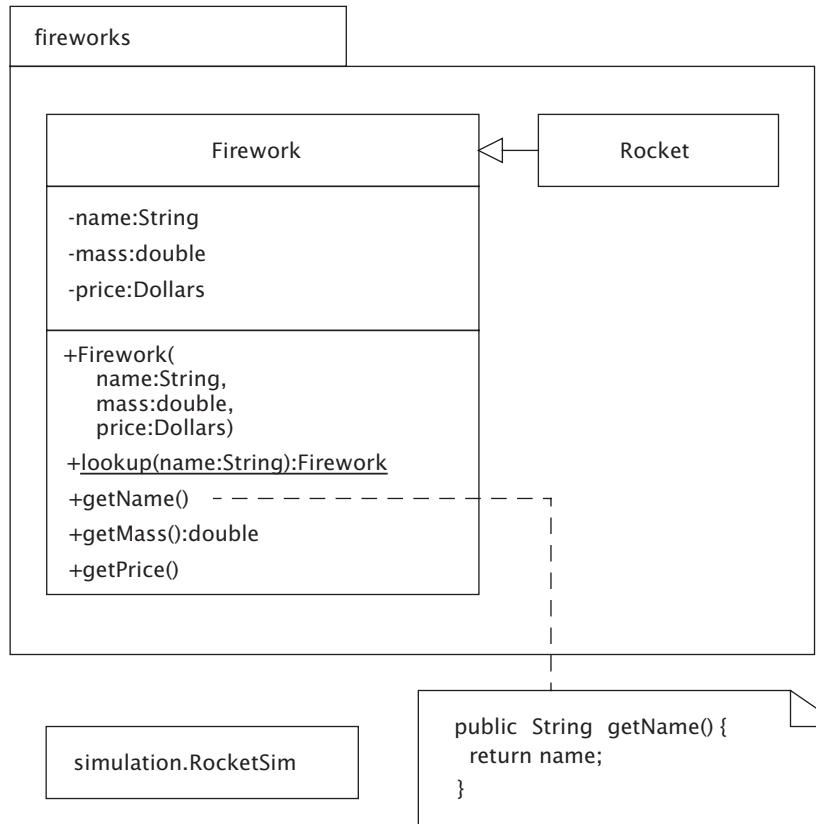


Figure D.1

Le package `Fireworks` inclut les classes `Firework` et `Rocket`.

Relations entre classes

La Figure D.2 illustre quelques-unes des fonctionnalités d'UML servant à modéliser les relations inter-classes.

Voici quelques directives de notation pour les relations de classes :

- Spécifiez un nom de classe ou un nom de méthode en italiques pour signifier que la classe ou la méthode est abstraite. Soulignez le nom pour indiquer qu'elle est statique.
- Pour indiquer une relation sous-classe/super-classe, utilisez une flèche à tête creuse pointant vers la super-classe.
- Utilisez une ligne entre deux classes pour indiquer que leurs instances partagent une quelconque relation. Très fréquemment, une telle ligne signifie qu'une classe possède une variable d'instance se référant à une autre classe. Par exemple, la classe `Machine` prévoit une variable d'instance pour conserver une référence à un objet `TubMediator`.
- Utilisez un losange pour indiquer qu'une instance d'une classe contient une collection d'instances d'une autre classe.
- Une flèche à tête ouverte indique la navigabilité. Elle permet de signaler qu'une classe possède une référence à une autre classe, la classe pointée par la flèche, mais que celle-ci ne possède pas de référence réciproque.
- L'indicateur de pluralité, tel que `0..1`, indique le nombre de relations pouvant apparaître entre des objets. Utilisez le symbole astérisque (*) pour signaler que zéro ou plusieurs instances d'une classe peuvent être reliées à d'autres instances d'une classe associée.
- Lorsqu'une méthode peut générer une exception, vous pouvez l'indiquer à l'aide d'une flèche en pointillé allant de la méthode vers la classe d'exception. Etiquetez la flèche avec l'expression `throw`.
- Utilisez une flèche en pointillé entre deux classes pour indiquer une dépendance n'employant pas de référence d'objet. Par exemple, la classe `Customer` s'appuie sur une méthode statique du moteur de recommandation `LikeMyStuff`.

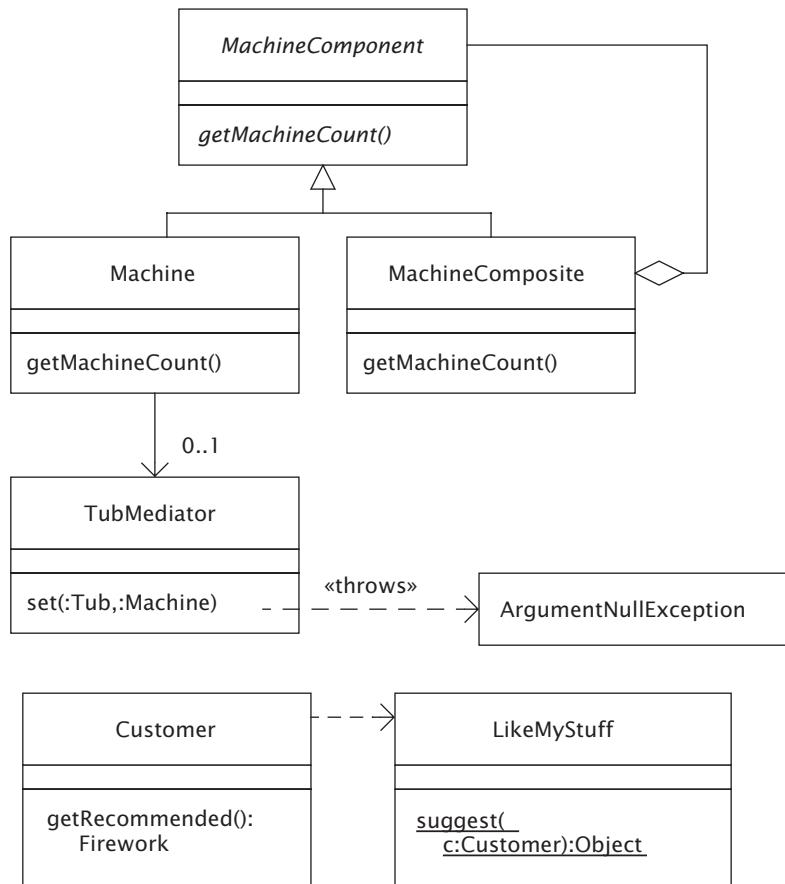


Figure D.2

Un objet `MachineComposite` contient des objets `Machine` ou d'autres objets composites. La classe `Customer` dépend de la classe `LikeMyStuff`.

Interfaces

La Figure D.3 illustre les fonctionnalités de base servant à la modélisation des interfaces.

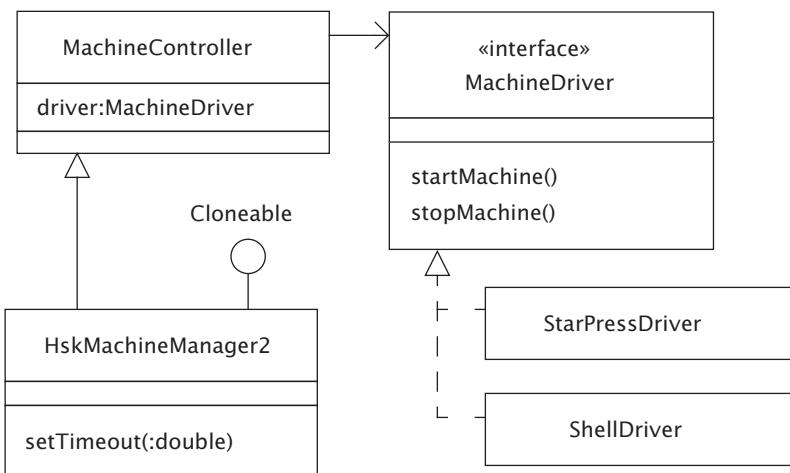


Figure D.3

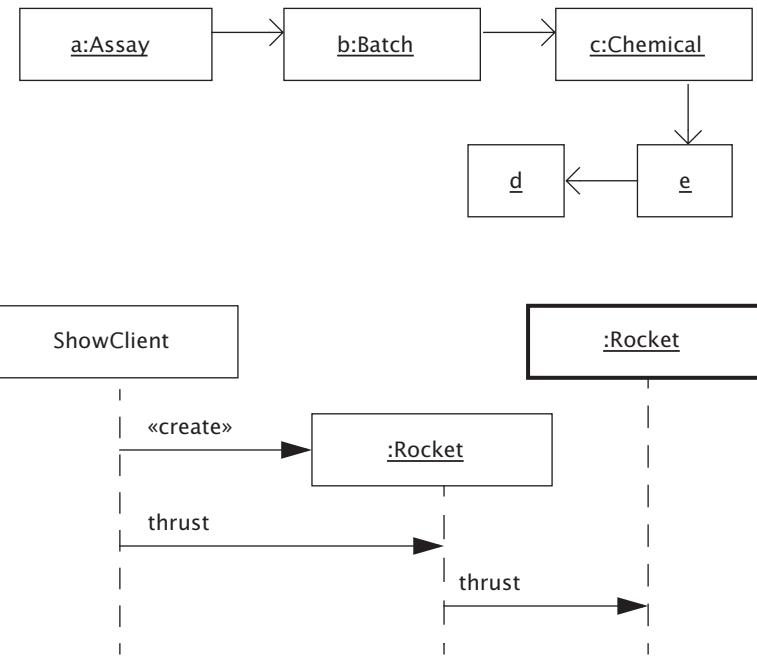
Vous pouvez signaler une interface au moyen d'une étiquette "interface" ou d'un symbole ressemblant à une sucette.

Voici quelques directives de représentation :

- Vous pouvez signaler une interface en plaçant dans un rectangle le texte "interface" et son nom, comme le montre la Figure D.3. Une flèche en pointillé à tête creuse permet d'indiquer qu'une classe implémente l'interface.
- Vous pouvez aussi signifier qu'une classe implémente une interface en utilisant une ligne surmontée d'un cercle (ressemblant à une sucette) avec le nom de l'interface.
- Les interfaces et leurs méthodes sont toujours abstraites en Java. Curieusement, elles n'apparaissent pas en italiques comme c'est le cas des classes abstraites et des méthodes abstraites spécifiées dans des classes.

Objets

La Figure D.4 illustre un exemple de diagramme d'objets servant à représenter des instances spécifiques de classes.

**Figure D.4**

Les représentations d'objets mentionnent leur nom et/ou leur type. Un diagramme de séquence signale une succession d'appels de méthodes.

Voici quelques directives de modélisation des objets :

- Spécifiez un objet en indiquant son nom et son type séparés par un signe double-point. Vous pouvez optionnellement n'indiquer que le nom ou que le signe double-point suivi du type. Dans tous les cas, soulignez le nom et/ou le type.
- Utilisez une ligne pour indiquer qu'un objet possède une référence à un autre objet. Vous pouvez utiliser une flèche à tête ouverte pour signaler la direction de la référence.
- Vous pouvez illustrer une séquence d'objets envoyant des messages, comme illustré dans la partie inférieure de la Figure D.4. L'ordre de lecture des messages est du haut vers le bas, et les lignes en pointillé indiquent l'existence de l'objet dans le temps.
- Utilisez l'étiquette "create" pour montrer qu'un objet en crée un autre. La Figure D.4 illustre la classe ShowClient créant un objet local Rocket.

- Encadrez un objet par une bordure épaisse pour signaler qu'il est actif dans un autre thread, processus ou ordinateur. La Figure D.4 montre un objet local Rocket transmettant une requête relative à sa méthode thrust(); à un objet Rocket s'exécutant sur un serveur.

Etats

La Figure D.5 illustre un diagramme d'états.

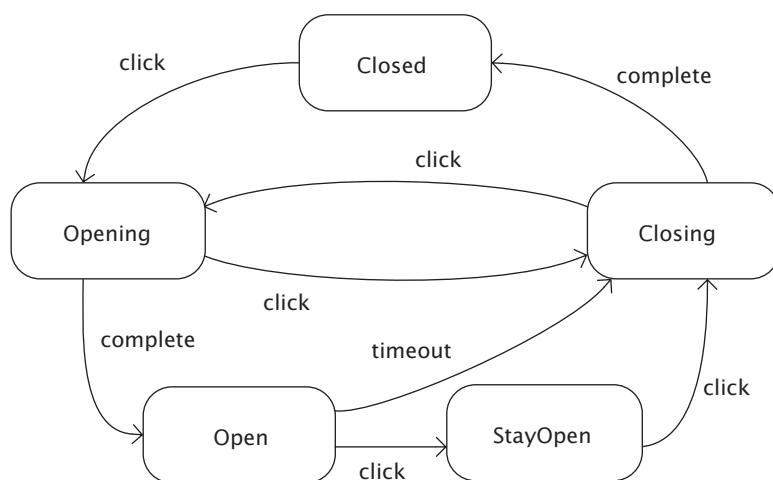


Figure D.5

Un diagramme d'états montre les transitions d'un état à l'autre.

Voici des directives concernant la modélisation d'états :

- Indiquez un état dans un rectangle à coins arrondis.
- Signalez une transition entre deux états au moyen d'un flèche à tête ouverte.
- Un diagramme d'états n'a pas besoin de correspondre directement à une classe ou un diagramme d'objets. Il peut toutefois représenter une transposition directe, comme le montre la Figure 22.3 du Chapitre 22.

Glossaire

Abstraction. Une classe qui dépend de méthodes abstraites implémentées dans des sous-classes ou dans les implementations d'une *interface*.

Adaptateur de classe. Un ADAPTER qui étend la classe à adapter et répond aux exigences de l'interface cible.

Adaptateur d'objet. Un ADAPTER qui étend une classe cible et délègue à une classe existante.

Algorithme. Une procédure de calcul bien définie qui reçoit un ensemble de valeurs en entrée et produit une valeur en sortie.

Analyse. L'analyse d'une préparation chimique.

Analyseur syntaxique. Un objet qui reconnaît les éléments d'un langage et décompose leur structure d'après un ensemble de règles pour les mettre dans une forme adaptée à un traitement subséquent.

API (*Application Programming Interface*). L'interface, ou l'ensemble d'appels, qu'un système rend accessible publiquement.

Apogée. Le point le plus élevé de la trajectoire d'un artifice.

Arbre. Un modèle objet qui ne contient pas de *cycles*.

Arbre syntaxique abstrait. Une structure, créée par un *analyseur syntaxique*, qui organise le texte en entrée d'après la grammaire d'un langage.

Bombe aérienne. Un artifice lancé à partir d'un *mortier* et qui explose en plein vol, éjectant des *étoiles* mises à feu.

Carrousel. Un grand rack intelligent qui accepte des produits par une porte et les stocke.

Chandelle romaine. Un tube stationnaire qui contient un mélange de charges explosives et d'étoiles.

Chemin. Dans un modèle objet, une série d'objets telle que chaque objet possède une référence vers l'objet suivant.

Client. Un objet qui utilise ou requiert les méthodes d'un autre objet.

Composite. Un groupe d'objets dans lequel certains objets peuvent en contenir d'autres, de façon que certains objets représentent des groupes et d'autres représentent des éléments individuels, ou *feuilles*.

Constructeur. Dans Java, une méthode spéciale dont le nom correspond à celui d'une classe et qui sert à instancier cette classe.

Copie complète. Une copie d'un objet dans laquelle les attributs du nouvel objet sont des copies complètes des attributs de l'original.

Copie partielle. Une copie d'un objet dans laquelle les attributs du nouvel objet ne sont pas des copies complètes des attributs de l'original, laissant le nouvel objet partager avec l'original des objets subordonnés.

CORBA (*Common Object Request Broker Architecture*). Une conception standard (architecture commune) qui supporte la transmission de requêtes d'objets entre systèmes.

Couche. Un groupe de classes possédant des responsabilités similaires, souvent réunies dans une bibliothèque, et présentant généralement des dépendances bien définies avec d'autres couches.

Couplage lâche. Une responsabilité mutuelle limitée et bien définie entre des objets qui interagissent.

Cycle. Un *chemin* le long duquel un nœud, ou objet, apparaît deux fois.

Double dispatching. Une conception dans laquelle un objet de classe B passe une requête à un objet de classe A, lequel la repasse immédiatement à l'objet de classe B, avec des informations additionnelles sur le type de l'objet de classe A.

Driver. Un objet qui opère sur un système informatique, tel qu'une base de données, ou sur un équipement externe, tel qu'un traceur, conformément à une interface bien définie.

EJB (Enterprise JavaBeans). Une spécification d'architecture multiniveau basée sur des composants.

Encapsulation. Une conception qui limite, au moyen d'une interface spécifique, l'accès aux données et aux opérations d'un objet.

Equations paramétriques. Des équations qui définissent un groupe de variables, telles que x et y , en tant que fonctions d'un paramètre standard, tel que t .

Etat. Une combinaison des valeurs courantes des attributs d'un objet.

Etoile. Une petite bille formée à partir d'un mélange explosif, entrant habituellement dans la composition d'une *bombe aérienne* ou d'une *chandelle romaine*.

Feuille. Un élément individuel dans un *composite*.

Flux. Une séquence d'octets ou de caractères, comme celles apparaissant dans un document.

Grammaire. Un ensemble de règles de composition.

Graphe. Un ensemble de nœuds et d'arêtes.

Graphe orienté. Un *graphe* dans lequel les arêtes ont une direction.

GUI (Graphical User Interface). Dans une application, une couche logicielle qui permet à l'utilisateur d'interagir avec des boutons, des menus, des barres de défilement, des zones de texte, et d'autres composants graphiques.

Hiérarchie parallèle. Une paire de hiérarchies de classes dans laquelle chaque classe d'une hiérarchie possède une classe correspondante dans l'autre hiérarchie.

Hook. Un appel de méthode placé par un développeur dans le code pour permettre à d'autres développeurs d'insérer du code à un point spécifique d'une procédure.

IDE (Integrated Development Environment). Un ensemble logiciel combinant des outils pour l'édition et le débogage de code et des outils pour la création de programmes.

Immutable. Qualifie un objet dont les valeurs ne peuvent pas changer.

Implémentation. Les instructions qui forment le corps des méthodes d'une classe.

Initialisation paresseuse. L'instanciation d'un objet seulement au moment où il est requis.

Interface. L'ensemble des méthodes et des champs d'une classe auxquels des objets d'autres classes sont autorisés à accéder. Egalement une interface Java qui définit les méthodes que la classe d'implémentation doit fournir.

Interface de marquage. Une interface qui ne déclare aucun champ ou méthode, dont la simple présence indique quelque chose. Par exemple, `Cloneable` est une interface de marquage qui garantit à ses implémentateurs qu'ils supporteront la méthode `clone()` définie dans `Object`.

Interface graphique utilisateur. Voir *GUI*.

Interpréteur. Un objet composé à partir d'une hiérarchie de composition dans laquelle chaque classe représente une règle de composition déterminant comment elle implémente ou interprète une opération qui survient dans la hiérarchie.

JDBC. Une interface de programmation d'applications pour l'exécution d'instructions SQL. JDBC est une marque, non un sigle.

JDK (*Java Development Kit*). Un ensemble logiciel incluant des bibliothèques de classes Java, un compilateur, et d'autres outils associés. Désigne souvent spécifiquement les kits disponibles à l'adresse java.sun.com.

JUnit. Un environnement de test, écrit par Erich Gamma et Kent Beck, qui permet d'implémenter des tests de régression automatisés dans Java. Disponible à l'adresse www.junit.org.

Kit. Une classe avec des méthodes de création qui retournent des instances d'une famille d'objets. Voyez le Chapitre 17, ABSTRACT FACTORY.

Langage de consolidation. Un langage informatique, tel que Java ou C#, qui tente de préserver les points forts de ses prédécesseurs et d'éliminer leurs faiblesses.

Loi de Demeter. Un principe de conception orienté objet qui stipule qu'une méthode d'un objet devrait envoyer des messages uniquement aux objets argument, à l'objet lui-même, ou aux attributs de l'objet.

Méthode. L'implémentation d'une *opération*.

Modèle-Vue-Contrôleur. Une conception qui sépare un objet intéressant, le modèle, des éléments de GUI qui le représentent et le manipulent, la vue et le contrôleur.

Mole. Par définition, c'est la quantité de matière d'un système contenant autant d'entités élémentaires qu'il y a d'atomes dans 12 grammes de carbone 12.

Ce nombre permet d'appliquer des équations chimiques tout en travaillant avec des quantités mesurables de préparations chimiques. Si m_w est le poids moléculaire d'une entité chimique, m_w grammes de cette entité chimique contiendra un mole de cette entité chimique.

Mortier. Un tube à partir duquel une bombe aérienne est lancée.

Multiniveau. Un type de système qui assigne des couches de responsabilités à des objets s'exécutant sur différents ordinateurs.

Mutex. Un objet partagé par des threads rivalisant pour obtenir le contrôle du *verrou* sur l'objet. Ce terme signifie littéralement *exclusion mutuelle*.

Niveau. Une couche logicielle qui s'exécute sur un ordinateur.

Objet métier. Un objet qui modélise une entité ou un processus dans une entreprise.

Oozinoz. Une entreprise fictive qui fabrique et vend des pièces pour feux d'artifices et organise des événements pyrotechniques.

Opération. La spécification d'un service qui peut être demandé à partir d'une instance d'une classe.

Pattern. Un moyen d'accomplir quelque chose, d'atteindre un objectif.

Pattern de conception. Un *pattern* qui opère approximativement au niveau classe.

Polymorphisme. Le principe selon lequel l'invocation d'une méthode dépend à la fois de l'opération invoquée et du récepteur de l'invocation.

Presse à étoiles. Une machine qui moule une préparation chimique en lui donnant la forme d'étoiles.

Principe de substitution de Liskov. Un principe de conception orienté objet qui stipule qu'une instance d'une classe devrait fonctionner comme une instance de sa super-classe.

Racine. Dans un *arbre*, un nœud ou objet distinct qui n'a pas de parent.

Refactorisation. La modification de code afin d'améliorer sa structure interne sans changer son comportement extérieur.

Réflexion. La possibilité de travailler avec des types et des membres de types en tant qu'objets.

Relation. Le rapport qu'entretiennent des objets. Dans un modèle objet, il s'agit du sous-ensemble de toutes les références possibles des objets d'un type à des objets d'un second type.

RMI (*Remote Method Invocation*). Une fonctionnalité Java qui permet à des objets situés sur des ordinateurs différents de communiquer.

Session. L'événement qui consiste pour un utilisateur à exécuter un programme, à accomplir des transactions dans ce programme, et à le quitter.

Signature. Une combinaison du nom d'une méthode ainsi que du nombre et du type de ses paramètres formels.

SQL (*Structured Query Language*). Un langage informatique pour interroger des bases de données relationnelles.

Stockage persistant. Une forme de stockage sur un matériel, tel qu'un disque, où les informations sont préservées même si la machine est arrêtée.

Stratégie. Un plan, ou une approche, pour atteindre un but selon certaines conditions initiales.

Théorie des graphes. Une conception mathématique de nœuds et d'arêtes. Lorsqu'elle est appliquée à un modèle objet, les nœuds du graphe sont généralement des objets et ses arêtes sont habituellement des références à ces objets.

Traversée post-ordonnée. Une itération sur un arbre ou un autre objet composite lors de laquelle un nœud est retourné après ses descendants.

Traversée pré-ordonnée. Une itération sur un arbre ou un autre objet composite lors de laquelle un nœud est retourné avant ses descendants.

Trémie. Un conteneur qui dispense des produits chimiques, généralement dans une machine.

Type de retour covariant. Lorsqu'une sous-classe remplace une méthode et déclare son type de retour comme étant une sous-classe du type de retour du parent.

UML (*Unified Modeling Language*). Une notation permettant de représenter des idées de conception.

URL (*Uniform Resource Locator*). Un pointeur vers une ressource Web.

Verrou. Une ressource exclusive qui représente la possession d'un objet par un thread.

WIP (*Work In Process*). Des biens en cours de fabrication dans une usine.

XML (*Extensible Markup Language*). Un langage textuel qui utilise des balises contenant des informations relatives au texte et qui sépare précisément les classes ou types de documents de leurs instances.

Bibliographie

Alexander, Christopher. 1979. *The Timeless Way of Building*. Oxford, England : Oxford University Press.

Alexander, Christopher, Sara Ishikawa, et Murray Silverstein. 1977. *A Pattern Language: Towns, Buildings, Construction*. Oxford, England : Oxford University Press.

Arnold, Ken, et James Gosling. 1998. *The JavaTM Programming Language, Second Edition*. Reading, MA : Addison-Wesley.

Booch, Grady, James Rumbaugh, et Ivar Jacobsen. 1999. *The Unified Modeling Language User Guide*. Reading, MA : Addison-Wesley.

Buschmann, Frank, et al. 1996. *Pattern-Oriented Software Architecture: A System of Patterns*. Chichester, West Sussex, England : John Wiley & Sons.

Cormen, Thomas H., Charles E. Leiserson, et Ronald L. Rivest. 1990. *Introduction to Algorithms*. Cambridge, MA : MIT Press.

Cunningham, Ward, ed. The Portland Patterns Repository. www.c2.com.

Flanagan, David. 2005. *JavaTM in a Nutshell*, 5th ed. Sebastopol, CA : O'Reilly & Associates.

Flanagan, David, Jim Farley, William Crawford, et Kris Magnusson. 2002. *JavaTM Enterprise in a Nutshell*, 2d ed. Sebastopol, CA : O'Reilly & Associates.

Fowler, Martin, Kent Beck, John Brant, William Opdyke, et Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Reading, MA : Addison-Wesley.

Fowler, Martin, et Kendall Scott. 2003. *UML Distilled, Third Edition*. Boston, MA : Addison-Wesley.

Gamma, Erich, Richard Helm, Ralph Johnson, et John Vlissides. *Design Patterns*. 1995. Boston, MA : Addison-Wesley.

- Gosling, James, Bill Joy, Guy Steele, et Gilad Bracha. 2005. *The Java™ Language Specification, Third Edition*. Boston, MA : Addison-Wesley.
- Kerievsky, Joshua. 2005. *Refactoring to Patterns*. Boston, MA : Addison-Wesley.
- Lea, Doug. 2000. *Concurrent Programming in Java™, Second Edition*. Boston, MA : Addison-Wesley.
- Lieberherr, Karl J., et Ian Holland. 1989. "Assuring good style for object-oriented programs". Washington, DC. IEEE Software.
- Liskov, Barbara. May 1987. "Data abstraction and hierarchy". *SIGPLAN Notices*, volume 23, number 5.
- Metsker, Steven J. 2001. *Building Parsers with Java™*. Boston, MA : Addison-Wesley.
- Russell, Michael S. 2000. *The Chemistry of Fireworks*. Cambridge, UK : Royal Society of Chemistry.
- Vlissides, John. 1998. *Pattern Hatching: Design Patterns Applied*. Reading, MA : Addison-Wesley.
- Wake, William C. 2004. *Refactoring Workbook*. Boston, MA : Addison-Wesley.
- Weast, Robert C., ed. 1983. *CRC Handbook of Chemistry and Physics*, 63rd ed. Boca Raton, FL : CRC Press.
- White, Seth, Mayderne Fisher, Rick Cattell, Graham Hamilton, et Mark Hapner. 1999. *JDBC™ API Tutorial and Reference, Second Edition*. Boston, MA : Addison-Wesley.
- Wolf, Bobby. 1998. "Null object", in *Pattern Languages of Program Design 3*, ed. Robert Martin, Dirk Riehle, et Frank Buschmann. Reading, MA : Addison-Wesley.

Index

A

ABSTRACT FACTORY
et FACTORY METHOD 178
et packages 182
kits de GUI 173
objectif 173
solutions aux exercices 379

Abstractions
BRIDGE 61
drivers 66

Adaptateurs
d'objets 25
de classes 25

ADAPTER
adaptateurs de classes et d'objets 25
identification 33
objectif 21
pour des données JTable 29
pour des interfaces 21
solutions aux exercices 338

Algorithmes 207
complétion 215
de tri 211
et méthodes 207
vs stratégies 235

Alternances
et VISITOR 327
vs séquences 56

Analyseurs syntaxiques
INTERPRETER (pour) 265
pour BUILDER 158
VISITOR (pour) 329

Annulation d'opérations 189

Applications orientées objet 35

Arbres
dans COMPOSITE 50
syntaxiques abstraits 329

Arêtes de graphes 50

Arrays (classe) 212

ASP.NET (Active Server Pages for .NET)
122

Attributs
changeants, objets 223
comparaison dans un algorithme de tri 212
d'un objet copié 185

B

Boucles
for 296
infinies 57

BRIDGE
abstraction 61
drivers 66
JDBC 67
objectif 61
solutions aux exercices 348

BufferedWriter (classe) 278

BUILDER
avec des contraintes 160
objectif 157
ordinaire 157
solutions aux exercices 373
tolérant 163

C**CHAIN OF RESPONSABILITY**

ancrege 140
 objectif 135
 ordinaire 135
 refactorisation (pour appliquer) 137
 sans COMPOSITE 142
 solutions aux exercices 364

Chaînes

analyse syntaxique 157
 immuables 144

Chargeur de classe, PROXY 128**Chemin, COMPOSITE 51****Classes**

adaptateurs (de) 25
 chargeur (de) 128
 constructeurs (de) 153
 de contrôle 62
 de filtrage 279
 de handshaking 64
 extension 269
 familles (de) 182
 hiérarchies 61
 hiérarchies parallèles 169
 implémentation 15
 instances uniques 79
 instantiation 167
 interfaces 15
 modélisation UML 409
 pour des couches de code 92
 stub 18
 visibilité 75

Classes abstraites

adaptateurs d'objets (pour des) 29
 dans des hiérarchies 61
 et interfaces 16
 pour des classes de filtrage 279

Clients

définition des exigences dans une interface 25
 en tant que listeners, Swing 86
 enregistrement pour notifications 193

et proxies 122

objets (en tant que) 21
 partage d'objets (entre) 143

Clones

de collections 302
 prototypage (avec des) 185

Cohérence relationnelle 107**Collections**

boucle for 296
 clones 302
 itérateurs 165, 297
 pour la sécurité inter-threads 297
 tri 212

Collections (classe) 212**COMMAND**

commandes de menus 245
 en relation avec d'autres patterns 251
 fourniture d'un service 248
 hooks 249
 objectif 245
 solutions aux exercices 393

Commandes de menus 245**Comparaisons**

conditions booléennes 262
 dans des algorithmes de tri 212

Comparator (interface) 212**Comportement récursif dans les composites 48****COMPOSITE**

arbres 50
 comportement récursif 48
 cycles 51, 55
 feuilles 47
 groupes d'objets 47
 itérateurs 303
 non-arbre 59
 objectif 47
 ordinaire 47
 solutions aux exercices 345
 traversées pré-ordonnée/post-ordonnée 303

- C**
- Concurrence, POO 81
 - Conditions booléennes 261
 - Consolidation, langage (de) 7
 - Constantes, déclaration dans une interface 19
 - Constructeurs de classes 153
 - Construction 153
 - avec des contraintes 160
 - défis 153
 - progressive d'objets 157
 - solutions aux exercices 371
 - Contraintes de construction 160
 - Contrat
 - d'une interface 17
 - entre les développeurs 221
 - Contrôleurs, conception MVC 90
 - Copie
 - complète 383
 - de collections 302
 - de prototypes 185
 - partielle 383
 - CORBA (Common Object Request Broker Architecture) 122
 - Corps de méthodes 204
 - Couches de code, conception MVC 92
 - Couplage lâche, MEDIATOR 105
 - Cycles
 - conséquences 59
 - dans COMPOSITE 51, 55
 - et VISITOR 323
- D**
- DECORATOR**
 - en relation avec d'autres patterns 292
 - enveloppeurs de fonctions 285
 - flux et objets d'écriture 277
 - objectif 277
 - solutions aux exercices 399
 - Découplage 19
 - Demeter, loi (de) 271
 - Démos 36, 342
 - Dépendances, OBSERVER 98
 - Diagramme de séquence UML 68
 - Double dispatching, VISITOR 322
 - Drivers
 - de base de données 67
 - en tant que BRIDGE 66
 - Durée de vie, MEMENTO 196
- E**
- Eclipse (IDE) 35
 - Encapsulation 77
 - En-têtes de méthodes 204
 - Enveloppeurs de fonctions 285
 - Environnement de développement intégré 35
 - Equations paramétriques 40
 - Erreurs potentielles, élimination 273
 - Etats
 - attributs changeants 223
 - constants 231
 - dans MEMENTO 189
 - machine (à) 225
 - modélisation 223, 416
 - Exceptions
 - déclaration 206
 - distantes 123
 - génération 206
 - gestion 206
 - Extensions 269
 - indicateurs 273
 - loi de Demeter 271
 - principe de substitution de Liskov 270
 - solutions aux exercices 398

F**FAÇADE**

- démos 36
- équations paramétriques 40
- objectif 35
- refactorisation (pour appliquer) 37
- solutions aux exercices 342
- utilitaires 36

FACTORY METHOD

- contrôle du choix de la classe à instancier 167
- dans une hiérarchie parallèle 169
- et ABSTRACT FACTORY 178
- identification 166
- itérateurs 165
- objectif 165
- solutions aux exercices 375

Familles

- d'objets 179
- de classes 182

Feuilles

- dans COMPOSITE 47
- énumération 311

FileWriter (classe) 278, 280**Filtres de flux** 279**Flux d'E/S** 277**FLYWEIGHT**

- objectif 143
- objets immuables 143
- partage d'objets 146
- refactorisation (pour appliquer) 144
- solutions aux exercices 368

Fonctions

- booléennes 404
- enveloppeurs (de) 285

for (boucle) 295**foreach (boucle)** 295**G****Grammaire d'un langage de programmation** 266**Graphes orientés** 51**GUI (Graphical User Interface)**

- conception Modèle-Vue-Contrôleur (MVC) 90
- et MEDIATOR 101
- et OBSERVER 85
- kits (de) 173
- pour des commandes de menus 245
- pour une application de visualisation 175

H**Handshaking, classes (de)** 64**Hiérarchies de classes** 61

- extensions 315
- parallèles 169
- stabilité 328

Hooks

- pour COMMAND 249
- pour TEMPLATE METHOD 218

I**IDE (Integrated Development Environment)** 35**Identification**

- de ADAPTER 33
- de FACTORY METHOD 166
- de SINGLETON 82

Immuabilité, FLYWEIGHT 143**Implémentations**

- d'opérations abstraites 61
- de classes 15
- par défaut 29
- vides 18

Indicateurs 273

Initialisation paresseuse

- de planificateurs 220
- de singletons 80

Instances

- initialisation paresseuse 80
- principe de substitution de Liskov 270
- uniques 79

Intégrité relationnelle 107**Interfaces**

- adaptation (à des) 21
- contrat 17
- de marquage 338
- et classes abstraites 16
- et proxies dynamiques 128
- Java 15
- modélisation UML 414
- obligations 17
- pour VISITOR 330

INTERPRETER

- exemple 254
- langages et analyseurs syntaxiques 265
- objectif 253
- solutions aux exercices 396

Inverse d'une relation binaire 107**InvocationHandler (interface)** 129**ITERATOR**

- ajout d'un niveau de profondeur à un énumérateur 310
- avec sécurité inter-threads 297
- énumération des feuilles 311
- objectif 295
- ordinaire 295
- pour un composite 303
- solutions aux exercices 401
- traversées pré-ordonnée/post-ordonnée 303

J**JavaBeans** 122**JDBC** 67**JDK (Java Development Kit)** 29**JTable (adaptation pour)** 29**JUnit** 248, 406**K****Kits**

- d'outils 35
- de GUI 173

L**Langage de consolidation** 7**Liskov, principe de substitution (de)** 270**Listeners, Swing** 86, 246**Loi de Demeter** 271**M****Machine à états UML** 225**Marquage, interfaces (de)** 338**MEDIATOR**

- objectif 101
- pour l'intégrité relationnelle 106
- pour les GUI 101
- refactorisation (pour appliquer) 104
- solutions aux exercices 359

MEMENTO

- annulation d'opérations 189
- durée de vie 196
- informations d'état 189
- objectif 189
- persistance entre les sessions 197
- solutions aux exercices 384

Menus

- commandes (de) 245
- Java 246

Méthodes 203

- constructeurs 153
- corps 204
- dans des classes stub 18

Méthodes (suite)

- en-tête 204
- et algorithmes 207
- et polymorphisme 208
- génération d'exceptions 206
- hooks 218, 249
- loi de Demeter 271
- minutage 248
- modèles 211
- modificateurs d'accès 204
- signatures 205
- utilitaires 39
- visibilité 75

Modèle objet 51

- relation (de) 107
- vs modèle relationnel 107

Modèle-Vue-Contrôleur (MVC)

- pour MEMENTO 191
- pour OBSERVER 90

Modélisation

- d'états 223, 416
- d'interfaces 414
- d'objets 414
- de classes 409
- de composites 60
- de conditions booléennes 261
- de relations inter-classes 412
- de stratégies 236
- UML 409

Modificateurs d'accès 76, 204**Mutex (objet)** 301**N****Noeuds**

- dans des graphes 50
- dans des itérations 303
- enfants 303
- parents 51
- têtes 303

O**Objets**

- adaptateurs (d') 25
- clients 21
- composites 47
- encapsulation 77
- état 223
- exécutables 253
- familles (d') 179
- feuilles 47
- immuables 143
- métiers 92
- modélisation UML 414
- mutex 301
- paires ordonnées 106
- partage 143
- racines 140
- références 51
- responsabilité 73, 79
- uniques 82
- verrous 81

Obligations des interfaces 17**Observable (classe)** 92**OBSERVER**

- approches push/pull 97
- dans les GUI 85
- maintenance 96
- Modèle-Vue-Contrôleur 90
- objectif 85
- refactorisation (pour appliquer) 88
- solutions aux exercices 354

Opérations

- abstraites 61
- annulation 189
- et méthodes 203
- signatures 205
- solutions aux exercices 387
- stratégiques 235

Outils, kits (d') 35

P**Packages**

- dans ABSTRACT FACTORY 182
- kits d'outils 35

Paires ordonnées d'objets 106**Paradigmes**

- classe/instance 7
- concurrents 7

Parents, nœuds 51**Partage d'objets** 143**Patterns** 3**Patterns de conception** 4

- classification 9
- construction 153
- extensions 269
- interfaces 15
- opérations 203
- responsabilité 73

Polymorphisme 208, 230, 245**Principe de substitution de Liskov** 270**Procédures** *Voir Algorithmes***Produit cartésien** 107**Programmation orientée aspect (POA)** 132**Programmation orientée objet (POO)**

- avec concurrence 81
- élimination des erreurs potentielles 273
- encapsulation 77
- loi de Demeter 271
- polymorphisme 208
- principe de substitution de Liskov 270

PROTOTYPE

- en tant qu'objet factory 183
- objectif 183
- pour des clones 185
- solutions aux exercices 382

PROXY

- distant 122
- dynamique 128

objectif 115

- pour des images 115
- solutions aux exercices 362
- suppression 120

Pull, OBSERVER 97**Push, OBSERVER** 97**R****Racines**

- dans CHAIN OF RESPONSABILITY 140
- dans des graphes 51

Refactorisation

- pour appliquer CHAIN OF RESPONSABILITY 137
- pour appliquer FACADE 37
- pour appliquer FLYWEIGHT 144
- pour appliquer MEDIATOR 104
- pour appliquer OBSERVER 88
- pour appliquer STATE 227
- pour appliquer STRATEGY 238
- pour appliquer TEMPLATE METHOD 219

Références d'objets 51**Réflexion**

- dans PROXY 129
- pour l'instanciation 154

Registre RMI 124**Relations entre objets** 106**repeat (boucle)** 295**Responsabilité**

- codage en couches 95
- contrôle par la visibilité 75
- couplage lâche 105
- distribution vs centralisation 77
- et encapsulation 77
- objets 79
- ordinaire 73
- solutions aux exercices 350

Réutilisabilité des kits d'outils 35**RMI (Remote Method Invocation)** 122

S**Sécurité inter-threads** 297**Séquences**

- d'instructions 211
- et VISITOR 327
- vs alternances 56

Services

- fourniture avec COMMAND 248
- spécification 203

Sessions 197**Signatures de méthodes** 205**SINGLETON**

- et threads 81
- identification 82
- mécanisme 79
- objectif 79
- solutions aux exercices 353

Solutions aux exercices

- ABSTRACT FACTORY 379
- ADAPTER 338
- BRIDGE 348
- BUILDER 373
- CHAIN OF RESPONSABILITY 364
- COMMAND 393
- COMPOSITE 345
- construction 371
- DECORATOR 399
- extensions 398
- FACADE 342
- FACTORY METHOD 375
- FLYWEIGHT 368
- interfaces 337
- INTERPRETER 396
- ITERATOR 401
- MEDIATOR 359
- MEMENTO 384
- OBSERVER 354
- opérations 387
- PROTOTYPE 382
- PROXY 362
- responsabilité 350
- SINGLETON 353
- STATE 390

STRATEGY 392

- TEMPLATE METHOD 389
- VISITOR 403

Sous-classes

- anonymes 94
- de classes stub 18
- pour des adaptateurs de classes 25
- pour INTERPRETER 257

STATE

- et STRATEGY 242
- états constants 231
- modélisation d'états 223
- objectif 223
- refactorisation (pour appliquer) 227
- solutions aux exercices 390

Stockage persistant 197**STRATEGY**

- et STATE 242
- et TEMPLATE METHOD 243
- modélisation de stratégies 236
- objectif 235
- refactorisation (pour appliquer) 238
- solutions aux exercices 392
- vs algorithmes 235

Stub, classes 18**Super-classes** 24**Swing** 245

- et OBSERVER 86
- listeners 246
- widget JTable 29

Systèmes multiniveaux 96**T****Tableaux**

- adaptation à une interface 29
- et boucles for 296
- tri 212

TEMPLATE METHOD

- algorithmes de tri 211
- complétion d'un algorithme 215
- et STRATEGY 243

hooks 218
 objectif 211
 refactorisation (pour appliquer) 219
 solutions aux exercices 389

Texte

analyse syntaxique 265
 filtres de mise en forme 280
 passage à la ligne 283

Théorie des graphes 51**Threads**

dans SINGLETON 81
 pour le chargement d'images 118
 sécurité (inter-) 297
 verrous 81

throws (clause) 204**Traversées pré-ordonnée/post-ordonnée 303****Tri, algorithmes (de) 211****Types de retour**

covariants 388
 de méthodes 205

notation 7

UnicastRemoteObject (interface) 123**Utilitaires, FACADE 36****V****Verrous**

dans une application multithread 301
 sur des objets 81

Visibilité

des classes et des méthodes 75
 et responsabilité 75

VISITOR

application (de) 315
 double dispatching 322
 et cycles 323
 inconvénients 328
 objectif 315
 ordinaire 318
 solutions aux exercices 403

Vues, conception MVC 90**U****UML (Unified Modeling Language) 409**

diagramme de séquence 68
 machine à états 225

W**while (boucle) 295****WIP (Work In Process) 82**

Les design patterns en Java

Les 23 modèles de conception fondamentaux

Tout programmeur Java se doit de connaître les 23 design patterns fondamentaux recensés par les célèbres développeurs du *Gang of Four*, véritable condensé de l'expérience de plusieurs générations de développeurs, et aujourd'hui incontournable pour écrire un code propre et efficace.

Cet ouvrage, fondé sur de nombreux exemples d'application, vous aidera à comprendre ces modèles et développera votre aptitude à les appliquer dans vos programmes.

Forts de leur expérience en tant qu'instructeurs et programmeurs Java, Steve Metsker et William Wake vous éclaireront sur chaque pattern, au moyen de programmes Java réels, de diagrammes UML, de conseils sur les bonnes pratiques et d'exercices clairs et pertinents. Vous passerez rapidement de la théorie à l'application en apprenant comment écrire un meilleur code ou restructurer du code existant pour le rationaliser, le rendre plus performant et plus facile à maintenir.

Code source disponible sur le site www.oozinoz.com !

À propos des auteurs :

Steven John Metsker, spécialiste des techniques orientées objet sous-jacentes à la création de logiciels épurés et performants, est l'auteur des ouvrages *Building Parsers with Java*, *Design Patterns Java Workbook* et *Design Patterns in C#* (parus chez Addison-Wesley). Il est décédé en février 2008.

William C. Wake (www.xp123.com) est consultant logiciel, coach et instructeur indépendant, avec plus de vingt années d'expérience en programmation. Il est l'auteur de *Refactoring Workbook* et *Extreme Programming Explored* (parus chez Addison-Wesley).

Programmation

Niveau : Avancé

Configuration : Multiplate-forme

PEARSON

Pearson Education France
47 bis, rue des Vinaigriers
75010 Paris
Tél. : 01 72 74 90 00
Fax : 01 42 05 22 17
www.pearson.fr

TABLE DES MATIÈRES

- Introduction aux interfaces
- ADAPTER
- FACADE
- COMPOSITE
- BRIDGE
- Introduction à la responsabilité
- SINGLETON
- OBSERVER
- MEDIATOR
- PROXY
- CHAIN OF RESPONSABILITY
- FLYWEIGHT
- Introduction à la construction
- BUILDER
- FACTORY METHOD
- ABSTRACT FACTORY
- PROTOTYPE
- MEMENTO
- Introduction aux opérations
- TEMPLATE METHOD
- STATE
- STRATEGY
- COMMAND
- INTERPRETER
- Introduction aux extensions
- DECORATOR
- ITERATOR
- VISITOR
- Code source d'Oozinoz
- Introduction à UML

ISBN : 978-2-7440-4097-9



9 782744 040979