

O'REILLY®

Second
Edition

Head First

Desi Patt

Édifice Ext.
& Entretien
Objet-Orien
Logiciel

**Éric Freeman et
Élisabeth Robso**

avec Kathy Sierra & Be



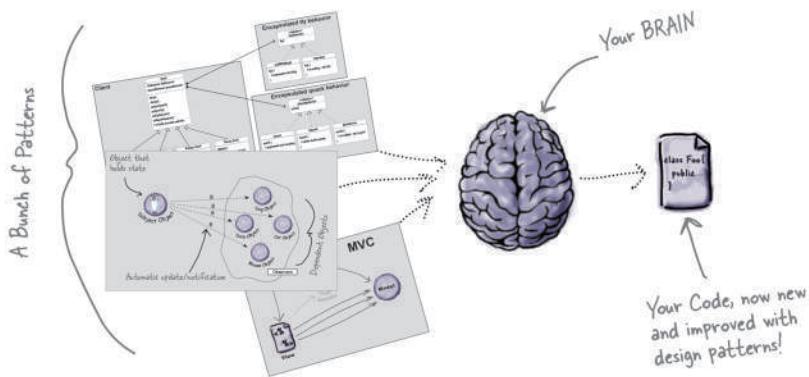
UN Cerveau-F

Head First

Design Patterns

What will you learn from this book?

You know you don't want to reinvent the wheel, so you look to Design Patterns—the lessons learned by those who've faced the same software design problems. With Design Patterns, you get to take advantage of the best practices and experience of others so you can spend your time on something more challenging. Something more fun. This book shows you the patterns that matter, when to use them and why, how to apply them to your own designs, and the object-oriented design principles on which they're based. Join hundreds of thousands of developers who've improved their object-oriented design skills through *Head First Design Patterns*.



What's so special about this book?

If you've read a Head First book, you know what to expect—a visually rich format designed for the way your brain works. With *Head First Design Patterns, 2E* you'll learn design principles and patterns in a way that won't put you to sleep, so you can get out there to solve software design problems and speak the language of patterns with others on your team.

COMPUTER PROGRAMMING

US \$69.99

CAN \$92.99

ISBN: 978-1-492-07800-5



5 6 9 9 9
9 781492 078005

"I received the book yesterday and started to read it...and I couldn't stop. This is très 'cool.' It is fun, but they cover a lot of ground and they are right to the point. I'm really impressed."

—**Erich Gamma**

IBM Distinguished Engineer, and coauthor of *Design Patterns*

"I feel like a thousand pounds of books have just been lifted off of my head."

—**Ward Cunningham**

inventor of the Wiki and founder of the Hillside Group

"*Head First Design Patterns* manages to mix fun, belly laughs, insight, technical depth, and great practical advice in one entertaining and thought-provoking read."

—**Richard Helm**

coauthor of *Design Patterns*

O'REILLY®

Éloge pour *Modèles de conception tête première*

« J'ai reçu le livre hier et j'ai commencé à le lire sur le chemin du retour... et je ne pouvais pas m'arrêter. Je l'ai amené à au gym et je m'attends à ce que les gens me voient beaucoup sourire pendant que je faisais de l'exercice et que je lisais. C'est vraiment "cool". C'est amusant, mais ils couvrent beaucoup de terrain et ils vont droit au but. Je suis vraiment impressionné.

— **Erich Gamma, ingénieur distingué IBM et coauteur de *Modèles de conception* avec le reste des Bande de quatre—Richard Helm, Ralph Johnson et John Vlissides**

"*Modèles de conception tête première* réussit à mélanger plaisir, rires de ventre, perspicacité, profondeur technique et grande pratique des conseils dans une lecture divertissante et stimulante. Que vous soyez nouveau dans les modèles de conception ou que vous les utilisiez depuis des années, vous êtes sûr d'obtenir quelque chose en visitant Objectville.

— **Richard Helm, coauteur de *Modèles de conception* avec le reste de la Bande des Quatre—Erich Gamma, Ralph Johnson et John Vlissides**

"J'ai l'impression que mille livres de livres viennent de m'être enlevés de la tête."

— **Ward Cunningham, inventeur du Wiki et fondateur du Hillside Group**

« Ce livre est proche de la perfection, de par la façon dont il allie expertise et lisibilité. Il parle avec autorité et se lit magnifiquement. C'est l'un des rares livres sur les logiciels que j'ai jamais lus et qui me semble indispensable. (Je mettrai peut-être 10 livres dans cette catégorie, à l'extérieur.) »

— **David Gelernter, professeur d'informatique à l'Université Yale et auteur de *Mondes miroirs et Beauté des machines***

« À Nose Plongez dans le royaume des motifs, un pays où les choses complexes deviennent simples, mais où les choses simples les choses peuvent aussi devenir complexes. Je ne vois pas de meilleurs guides touristiques qu'Éric et Elisabeth.

— **Miko Matsumura, analyste industriel, The Middleware Company, ancien évangéliste Java en chef, Sun Microsystems**

"J'ai ri, j'ai pleuré, ça m'a ému."

— **Daniel Steinberg, rédacteur en chef, java.net**

« Ma première réaction a été de me rouler par terre en riant. Après m'être relevé, j'ai réalisé que non seulement le livre techniquement précis, c'est l'introduction aux modèles de conception la plus simple à comprendre que j'ai vue.

— **Dr Timothy A. Budd, professeur agrégé d'informatique à l'Université d'État de l'Oregon et auteur de plus d'une douzaine de livres, dont *C++ pour les programmeurs Java***

« Jerry Rice gère mieux que n'importe quel receveur de la NFL, mais Éric et Elisabeth l'ont devancé.

Sérieusement... c'est l'un des livres les plus drôles et les plus intelligents sur la conception de logiciels que j'aie jamais lu.

— **Aaron LaBerge, vice-président principal de la technologie et du développement de produits, ESPN**

Plus d'éloges pour *Modèles de conception tête première*

« Une bonne conception de code est avant tout une bonne conception de l'information. Un concepteur de code enseigne à un ordinateur comment faire quelque chose, et il n'est pas surprenant qu'un grand professeur d'informatique se révèle être un grand professeur de programmeurs. La clarté admirable, l'humour et les doses substantielles d'intelligence de ce livre en font le genre de livre qui aide même les non-programmeurs à bien réfléchir à la résolution de problèmes.

— **Cory Doctorow, coéditeur de *Boing Boing*
et auteur de *En bas et en dehors dans le royaume magique*
et *Quelqu'un vient en ville, quelqu'un quitte la ville***

« Il y a un vieux dicton dans l'industrie de l'informatique et des jeux vidéo : eh bien, il ne peut pas être si vieux, car le la discipline n'est pas si ancienne – et elle ressemble à ceci : Le design, c'est la vie. Ce qui est particulièrement curieux à propos de cette expression, c'est que même aujourd'hui, presque personne qui travaille dans le domaine de la création de jeux électroniques ne peut s'entendre sur ce que signifie « concevoir » un jeu. Le concepteur est-il un ingénieur logiciel ? Un directeur artistique ? Un conteur ? Un architecte ou un constructeur ? Un pitch perso ou un visionnaire ? Un individu peut-il effectivement faire partie de tout ça ? Et surtout, qui s'en soucie, %\$!#&* ?

Il a été dit que le crédit « conçu par » dans le divertissement interactif s'apparente au crédit « réalisé par » dans le cinéma, ce qui lui permet en fait de partager son ADN avec peut-être le plus controversé, le plus exagéré et trop souvent dépourvu d'humilité. Bonne compagnie, hein ? Pourtant, si le design est la vie, il est peut-être temps de consacrer des cycles de qualité à réfléchir à ce que c'est.

Eric Freeman et Elisabeth Robson se sont courageusement portés volontaires pour regarder derrière le rideau du code pour nous *Modèles de conception tête première*. Je ne suis pas certain qu'aucun d'eux ne se soucie autant de la PlayStation ou de la Xbox, et ils ne devraient pas non plus le faire. Pourtant, ils abordent la notion de design à un niveau particulièrement honnête, de sorte que quiconque cherche à renforcer son ego pour renforcer sa brillante qualité d'auteur a intérêt à ne pas aller creuser ici où la vérité est révélée de manière stupéfiante. Les sophistes et les aboyeurs de cirque n'ont pas besoin de postuler. Littérateurs de la nouvelle génération, s'il vous plaît, venez munis d'un crayon.

— **Ken Goldstein, vice-président exécutif et directeur général,
Disney Online**

« C'est un texte de présentation difficile à écrire pour moi parce qu'Éric et Elisabeth étaient mes étudiants il y a longtemps, donc je ne veux pas paraître trop baveux, mais c'est le meilleur livre sur les modèles de design disponible pour les étudiants. Comme preuve : je l'utilise depuis sa publication, tant dans mes cours de deuxième cycle que de premier cycle, tant en génie logiciel qu'en programmation avancée. Dès sa sortie, j'ai laissé tomber le Gang of Four ainsi que tous les compétiteurs !

— **Gregory Rawlins, Université de l'Indiana**

« Ce livre combine la bonne humeur, de bons exemples et une connaissance approfondie des Design Patterns dans d'une manière qui rend l'apprentissage amusant. Étant dans l'industrie des technologies du divertissement, je suis intrigué par le principe d'Hollywood et le modèle de façade du cinéma maison, pour n'en nommer que quelques-uns. La compréhension des modèles de conception nous aide non seulement à créer des logiciels de qualité réutilisables et maintenables, mais nous aide également à perfectionner nos compétences en résolution de problèmes dans tous les domaines de problèmes. Ce livre est une lecture incontournable pour tous les professionnels et étudiants en informatique.

— **Newton Lee, fondateur et rédacteur en chef de l'Association for Computing
Machinery (ACM) Computers in Entertainment (acmcie.org)**

Éloge pour d'autres livres d'Eric Freeman et Elisabeth Robson

«J'adore vraiment ce livre. En fait, j'ai embrassé ce livre devant ma femme.

— **Satish Kumar**

"*Tête d'abord HTML et CSS*est une introduction résolument moderne aux pratiques avant-gardistes en matière de balisage et de présentation des pages Web. Il anticipe correctement les perplexités des lecteurs et les traite juste à temps. L'approche hautement graphique et incrémentale imite précisément la meilleure façon d'apprendre ce genre de choses : effectuez une petite modification et voyez-la dans le navigateur pour comprendre la signification de chaque nouvel élément. »

— **Danny Goodman, auteur de *HTML dynamique : le guide définitif***

« Le Web serait bien meilleur si tous les auteurs HTML commençaient par lire ce livre. »

— **L. David Baron, responsable technique, mise en page et CSS, Mozilla Corporation**
<http://dbaron.org>

« Ma femme a volé le livre. Elle n'a jamais fait de conception de sites Web, alors elle avait besoin d'un livre comme *Tête d'abord HTML et CSS*pour l'emmener du début à la fin. Elle a maintenant une liste de sites Web qu'elle souhaite créer – pour la classe de notre fils, notre famille... Si j'ai de la chance, je récupérerai le livre quand elle aura terminé.

— **David Kaminsky, maître inventeur, IBM**

Ce livre vous amène dans les coulisses de JavaScript et vous laisse une compréhension approfondie de comment fonctionne ce remarquable langage de programmation. J'aurais aimé avoir *Programmation JavaScript tête première* quand j'commençais !

— **Chris Fuselier, ingénieur-conseil**

"Le *La tête première*La série utilise des éléments de la théorie moderne de l'apprentissage, y compris le constructivisme, pour apporter lecteurs rapidement informés. Les auteurs ont prouvé avec ce livre que le contenu de niveau expert peut être enseigné rapidement et efficacement. Ne vous méprenez pas, c'est un livre JavaScript sérieux, et pourtant, une lecture amusante !

— **Frank Moore, concepteur et développeur Web**

Vous cherchez un livre qui vous intéressera (et vous fera rire) mais qui vous apprendra de la programmation sérieuse compétences? *Programmation JavaScript tête première*c'est ça !

— **Tim Williams, entrepreneur en logiciels**

Autres livres O'Reilly d'Eric Freeman et Elisabeth Robson

Tête d'abord, apprenez à coder

Programmation JavaScript tête première

Tête d'abord HTML et CSS

Programmation HTML5 en tête-à-tête

Autres livres connexes de O'Reilly

Java tête première

Apprendre Java

Java en bref

Entreprise Java en bref

Exemples Java en un mot

Livre de recettes Java

Modèles de conception J2EE

La tête première

Modèles de conception



Éric Freeman
Elisabeth Robson

O'REILLY®

Pékin • Boston • Farnham • Sébastopol • Tokyo

Modèles de conception Head First, 2e édition

par Eric Freeman, Elisabeth Robson, Kathy Sierra et Bert Bates

Droits d'auteur © 2021 Eric Freeman et Elisabeth Robson. Tous droits réservés.

Imprimé au Canada.

Publié par O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sébastopol, CA 95472.

Les livres O'Reilly Media peuvent être achetés à des fins éducatives, commerciales ou promotionnelles. Les éditions en ligne sont aussi offertes pour la plupart des titres (oreilly.com). Pour plus d'informations, contactez notre service des ventes corporatives/institutionnelles : (800) 998-9938 ou entreprise@oreilly.com.

Éditeurs 1re édition : Mike Hendrickson, Mike Loukides

Éditeurs 2e édition : Michele Cronin, Mélissa Duffield

Concepteur de la couverture : Ellie Volckhausen

Modèles de Wranglers : Éric Freeman, Elisabeth Robson

Historique d'impression :

Octobre 2004 : Première édition

Décembre 2020 : Deuxième édition

Historique des versions :

2020-11-10 Première version

Le logo O'Reilly est une marque déposée d'O'Reilly Media, Inc. Java et toutes les marques et logos basés sur Java sont des marques de commerce ou des marques déposées de Sun Microsystems, Inc. aux États-Unis et dans d'autres pays. O'Reilly Media, Inc. est indépendant de Sun Microsystems.

De nombreuses désignations utilisées par les fabricants et les vendeurs pour distinguer leurs produits sont revendiquées comme des marques.

Lorsque ces désignations apparaissent dans ce livre et que O'Reilly Media, Inc. était au courant d'une revendication de marque, les désignations ont été imprimées en majuscules ou en majuscules initiales.

Bien que toutes les précautions aient été prises lors de la préparation de ce livre, l'éditeur et les auteurs n'assument aucune responsabilité pour les erreurs ou omissions, ni pour les dommages résultant de l'utilisation des informations contenues dans ce document.

Autrement dit, si vous utilisez quelque chose dans *Modèles de conception tête première*, par exemple, gérer une centrale nucléaire, vous êtes seul. Nous vous encourageons cependant à utiliser l'application DJ View.

Aucun canard n'a été blessé lors de la confection de ce livre.

Le GoF original a accepté que ses photos soient dans ce livre. Oui, ils sont vraiment si beaux.

ISBN : 978-1-492-07800-5

[MBP]



Au Gang of Four, dont la perspicacité et l'expertise dans la saisie et la communication des modèles de conception ont changé à jamais le visage de la conception logicielle et amélioré la vie des développeurs du monde entier.

Maissérieusement, quand va-t-on voir une deuxième édition ?

Après tout, ça fait seulement *dix ans*.

vingt-cinq

Leauteurs

Auteurs de modèles de conception Head First



Eric Freeman
←



Elisabeth Robson
↓

Eric est décrit par Kathy Sierra, co-créatrice de la série Head First, comme « l'une de ces rares personnes maîtrisant la langue, la pratique et la culture de plusieurs domaines, du hackster hipster au vice-président d'entreprise, en passant par l'ingénieur et le groupe de réflexion ».

De formation, Éric est informaticien et a obtenu son doctorat à l'Université Yale.

Professionnellement, Éric était auparavant directeur technique de Disney Online et Disney.com chez Walt Disney Company.

Eric codirige maintenant la série Head First et consacre son temps à la création de contenu imprimé et vidéo chez WickedlySmart, qui est distribué sur les principales chaînes éducatives.

Les titres d'Eric's Head First incluent *Modèles de conception tête première*, *Tête d'abord HTML et CSS*, *Programmation JavaScript tête première*, *Programmation HTML5 en tête-à-tête*, et *Tête d'abord, apprenez à coder*.

Éric habite à Austin, au Texas.

Elisabeth est ingénieur logiciel, écrivain et formateur. Elle est passionnée par la technologie depuis qu'elle était étudiante à l'Université de Yale, où elle a obtenu une maîtrise ès sciences en informatique.

Elle est actuellement cofondatrice de WickedlySmart, où elle crée des livres, des articles, des vidéos et bien plus encore. Auparavant, en tant que directrice des projets spéciaux chez O'Reilly Media, Elisabeth a produit des ateliers en personne et des cours en ligne sur une variété de sujets techniques et a développé sa passion pour la création d'expériences d'apprentissage pour aider les gens à comprendre la technologie.

Lorsqu'elle n'est pas devant son ordinateur, Elisabeth fait de la randonnée, du vélo, du kayak et du jardinage en plein air, souvent avec son appareil photo à proximité.

Créateurs de la série Head First

Kathy Sierra



Bert Bates



Kathys'intéresse à l'apprentissage de la théorie depuis qu'elle était conceptrice de jeux pour Virgin, MGM et Amblin', et professeure de création de nouveaux médias à l'UCLA. Elle a été formatrice Java pour Sun Microsystems et a fondé JavaRanch.com (maintenant CodeRanch.com), qui a remporté les prix Jolt Cola Productivity en 2003 et 2004.

En 2015, elle a remporté le prix Pioneer de l'Electronic Frontier Foundation pour son travail de création d'utilisateurs compétents et de construction de communautés durables.

Kathy s'est récemment concentrée sur le coaching de pointe en matière de science du mouvement et d'acquisition de compétences, connu sous le nom de dynamique écologique ou « Éco-D ». Son travail utilisant Eco-D pour entraîner les chevaux ouvre la voie à une approche beaucoup plus humaine de l'équitation, provoquant le plaisir de certains (et malheureusement la consternation pour d'autres). Les chevaux chanceux (autonomes !) dont les propriétaires utilisent l'approche de Kathy sont plus heureux, en meilleure santé et plus athlétiques que leurs congénères entraînés traditionnellement.

Vous pouvez suivre Kathy sur Instagram :
@pantherflows.

Avant**Bert**t'était auteur, il était développeur, spécialisé dans l'IA à l'ancienne (principalement des systèmes experts), les systèmes d'exploitation en temps réel et les systèmes de planification complexes.

En 2003, Bert et Kathy ont écrit *Java tête première* et a lancé la série Head First. Depuis lors, il a écrit plus de livres sur Java et a consulté Sun Microsystems et Oracle sur plusieurs de leurs certifications Java. Il a aussi formé des centaines d'auteurs et d'éditeurs pour qu'ils créent des livres bien pédagogiques.

Bert est un joueur de Go et, en 2016, il a regardé avec horreur et fascination AlphaGo battre Lee Sedol. Depuis peu, il utilise Éco-D (dynamique écologique) pour améliorer son jeu de golf et entraîner son perroquet Bokeh.

Bert et Kathy ont eu le privilège de connaître Beth et Eric depuis maintenant 16 ans, et la série Head First est extrêmement chanceuse de les compter comme contributeurs clés.

Vous pouvez envoyer un message à Bert sur CodeRanch.com.

Table des matières (résumé)

	Introduction	xxv
1	Bienvenue dans les modèles de conception : <i>introduction aux modèles de conception</i> Gardez vos objets au courant : <i>le modèle d'observateur</i> Objets décoratifs : <i>le modèle de décorateur</i> Cuisiner avec OO Goodness : <i>le modèle d'usine</i> Objets uniques : <i>le modèle Singleton</i> Invocation d'encapsulation : <i>le modèle de commande</i> Être adaptatif : <i>les modèles d'adaptateur et de façade</i> Algorithmes d'encapsulation : <i>Modèle de méthode theTemplate</i> Collections bien gérées : <i>les modèles itérateur et composite</i> L'état des choses : <i>le modèle d'État</i> Contrôle de l'accès aux objets : <i>le modèle de proxy</i>	1 37 79 109 169 191 237 277 317 381 425 493 563 597
10	Modèles de modèles : <i>motifs composés</i> Modèles dans le monde réel : <i>mieux vivre avec des schémas</i> Annexe : <i>Modèles restants</i>	
11		
12		
13		
14		

Table des matières (la vraie affaire)

Introduction

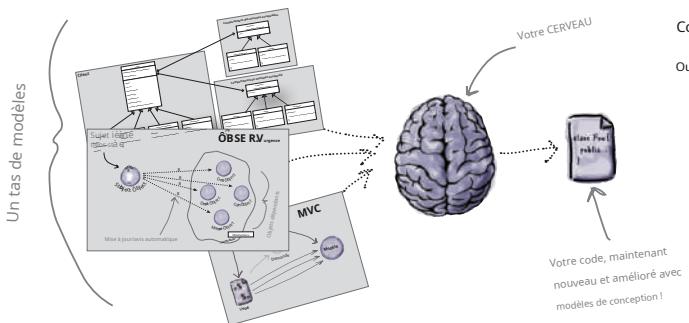
Votre cerveau sur les modèles de conception. Ici tu essaient de apprendre quelque chose, pendant qu'ici ton cerveau vous rend service en vous assurant que l'apprentissage ne se déroule pas bâton. Votre cerveau pense : « Mieux vaut laisser de la place à des choses plus importantes, comme les animaux sauvages à éviter et si la planche à neige nue est une mauvaise idée. » Alors comment fait vous faites croire à votre cerveau que votre vie dépend de la connaissance des modèles de design ?

À qui s'adresse ce livre ?	xxvi
On sait ce que vous pensez.	xxvii
Et on sait ce que pense ton cerveau. Nous considérons le lecteur « tête première » comme un apprenant.	xxviii
Métacognition : penser à penser Voici ce que NOUS avons fait	XXIX
Voici ce que VOUS pouvez faire pour plier votre cerveau à la soumission.	xxx
Réviseurs techniques	xxxii
Remerciements	xxxiv
	xxxv

1 Bienvenue dans les modèles de conception

Quelqu'un a déjà réglé vos problèmes. Dans ce chapitre, vous apprendrez pourquoi (et comment) vous pouvez exploiter la sagesse et les leçons apprises par d'autres développeurs qui ont suivi le même chemin de problèmes de conception et ont survécu au voyage. Avant d'avoir terminé, nous examinerons l'utilisation et les avantages des modèles de conception, examinerons quelques principes clés de conception orientée objet (OO) et passerons en revue un exemple du fonctionnement d'un modèle. La meilleure façon d'utiliser les modèles est de *charge ton cerveau avec eux et puis reconnaître des endroits dans vos conceptions et applications existantes où vous pouvez appliquer-les*. Au lieu de *decode* réutiliser, avec les modèles que vous obtenez *expériencer* utilisation.

Souviens-toi, sachant des concepts comme l'abstraction, l'héritage et le polymorphisme ne font pas de vous un bon orienté objet designer. Un gourou du design pense sur la façon de créer des conceptions flexibles et faciles à entretenir et qui peut faire face à changer.



Tout a commencé avec une simple application SimUDuck. Mais maintenant, on a besoin que les canards volent.	2
	3
	4
	5
Comment à propos d'une interface ? Quel serait <i>tusi</i> t'étais Joe ? La seule constante du développement logiciel Se concentrer sur le problème...	6
	7
	8
	9
Séparer ce qui change de ce qui reste le même Concevoir les comportements des canards	10
	11
Mettre en œuvre les comportements de Duck Intégrer le comportement de Duck	13
	15
Tester le code de Duck	18
Définir un comportement de manière dynamique	20
Aperçu des comportements encapsulés	22
HAS-A peut être meilleur que IS-A	23
En parlant de modèles de conception... entendus au restaurant local... entendus dans la cabine suivante...	24
	26
	27
La puissance d'un vocabulaire de modèles partagé	28
Comment utiliser les modèles de conception ?	29
Outils pour votre trousse d'outils de conception	32

le modèle d'observateur

2 Garder vos affaires au courant

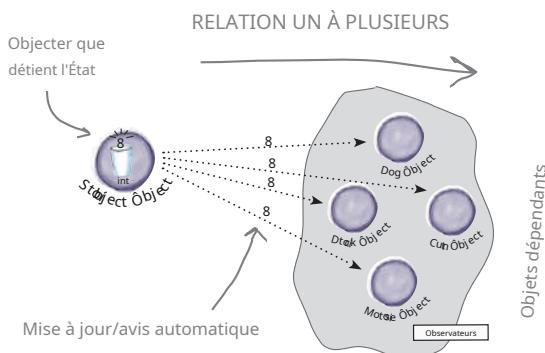
Vous ne voulez pas manquer quand quelque chose il se passe des affaires intéressantes, n'est-ce pas ?

Nous avons un modèle

qui garde vos objets *au courant* quand quelque chose il se soucie de ce qui se passe. C'est le modèle d'observateur. C'est l'un des modèles de conception les plus couramment utilisés et il est incroyablement utile. Nous allons examiner toutes sortes d'aspects intéressants d'Observer, comme son *relations un à plusieurs et accouplement lâche*. Et, avec ces concepts en tête, comment pouvez-vous ne pas être la vie de la fête des Patterns ?



Aperçu de l'application de surveillance météorologique	39
Découvrez le modèle d'observateur	44
Éditeurs + Abonnés = Modèle d'observateur	45
Le modèle d'observateur défini	51
La puissance du couplage lâche	54
Conception de la station météo Mise en œuvre de la station météo Mise sous tension de la station météo	57
À la recherche du modèle d'observateur dans la nature.	61
	65
	66
Testez les nouveaux outils de code pour votre boîte à outils de conception Défi des principes de conception	69
	71
	72
	73

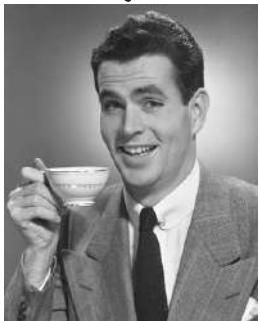
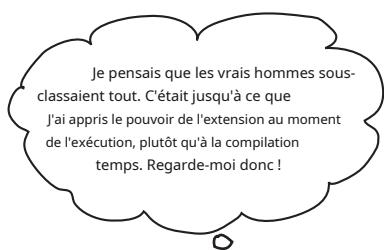


le modèle de décorateur

3 Objets décoratifs

Appelez simplement ce chapitre « Design Eye for the Inheritance »

Gars."Nous réexaminerons l'utilisation excessive typique de l'héritage et vous apprendrez à décorer vos classes au moment de l'exécution à l'aide d'une forme de composition d'objets. Pourquoi? Une fois que vous connaîtrez les techniques de décoration, vous serez en mesure de confier de nouvelles responsabilités à vos objets (ou à ceux de quelqu'un d'autre).*sans apporter de modifications au code des classes sous-jacentes.*



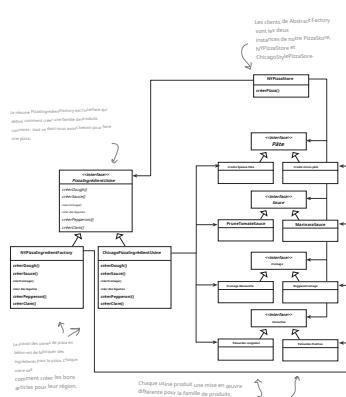
Bienvenue chez Starbuzz Coffee Le principe ouvert-fermé	80
Rencontrez le modèle du décorateur Construire une commande de boissons avec les décorateurs	86
Le modèle du décorateur défini	88
Décorer nos boissons	91
Écrire le code Starbuzz	92
Codifier les boissons	95
Codage des condiments	96
Servir quelques cafés	97
Décorateurs du monde réel : E/S Java Décorer les classes java.io	100
Écrire votre propre décorateur d'E/S Java Testez vos nouveaux outils de décoration d'E/S Java pour votre boîte à outils de conception	101
	102
	103
	105

le modèle d'usine

4 Cuisiner avec OO Goodness

Préparez-vous à créer des conceptions OO faiblement couplées.

Fabriquer des objets ne se limite pas à utiliser `lenouveauopérateur`. Vous apprenez que l'instanciation est une activité qui ne devrait pas toujours être pratiquée en public et qui peut souvent mener à *problèmes de couplage*. Et on veut pasce, n'est-ce pas ? Découvrez comment Factory Patterns peuvent vous aider à éviter des dépendances embarrassantes.



Identifier les aspects qui varient	112
Encapsuler la création d'objets	114
Construire une usine à pizza simple	115
La Simple Factory définie	117
Un cadre pour la pizzeria Permettre aux sous-classes de décider	120
Déclarer une méthode de fabrication	121
Il est enfin temps de rencontrer les créateurs et les produits de visualisation de modèles de méthodes d'usine en parallèle.	125
Modèle de méthode d'usine défini Examen	131
des dépendances d'objets Le principe	132
d'inversion des dépendances Application du principe	134
Familles d'ingrédients... Construire les usines d'ingrédients Retravailler	138
les pizzas...	146
Revisiter nos pizzerias	149
Qu'avons-nous fait ?	152
Modèle d'usine abstrait défini	153
Méthode d'usine et outils comparés Abstract Factory pour votre boîte à outils de conception	156
	160
	162

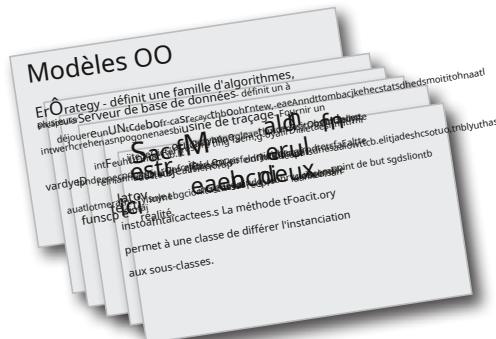
le modèle Singleton

5 Objets uniques

Notre prochain arrêt est le Singleton Pattern, notre billet pour créer des objets uniques en leur genre pour lesquels il n'existe que un exemple, jamais.

Vous serez peut-être heureux de savoir que de tous les modèles, le Singleton est le plus simple en termes de diagramme de classes ; en fait, le diagramme ne contient qu'une seule classe ! Mais ne soyez pas trop à l'aise ; malgré sa simplicité du point de vue de la conception de classe, sa mise en œuvre nécessitera une réflexion approfondie orientée objet. Alors mettez cette casquette de réflexion et allons-y.

Disséquer l'implémentation classique du modèle Singleton	173
The Chocolate Factory	175
Hershey, Pennsylvania	
Singleton Pattern défini Houston, nous avons un problème	177
Gérer le multithreading Peut-on améliorer le multithreading ? Pendant ce temps, de retour à la Chocolaterie... Des outils pour votre boîte à outils de conception	178
	180
	181
	183
	186



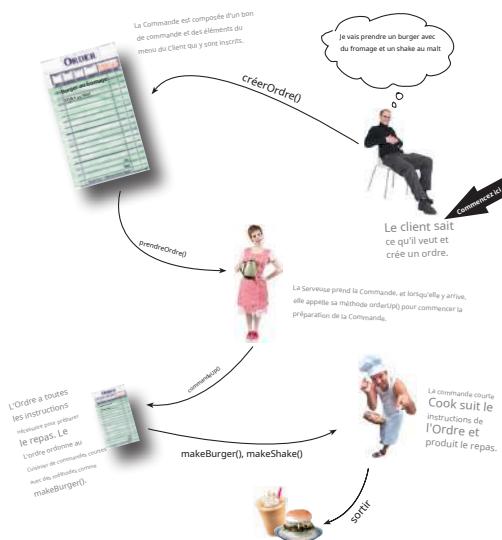
le modèle de commande

6

Invocation d'encapsulation

Dans ce chapitre, on amène l'encapsulation à un tout autre niveau : on va encapsuler l'invocation de méthode.

C'est vrai : en encapsulant l'invocation de méthode, on peut cristalliser des éléments de calcul afin que l'objet qui invoque le calcul n'ait pas à se soucier de la façon de faire les choses, il utilise simplement notre méthode cristallisée pour le faire. Nous pouvons également faire des choses extrêmement intelligentes avec ces invocations de méthodes encapsulées, comme les enregistrer pour la journalisation ou les réutiliser pour implémenter la fonctionnalité d'annulation dans notre code.



Domotique ou buste Consultez les classes des fournisseurs	192
	194
Une brève introduction au modèle de commande	197
Du Diner au modèle de commande Notre premier objet de commande	201
	203
Utilisation de l'objet de commande Affectation de commandes aux emplacements Mise en œuvre de la télécommande Mise en œuvre des commandes	204
	209
	210
	211
Mettre la télécommande à l'épreuve Il est temps d'écrire cette documentation... Qu'est-ce qu'on fait ?	212
	215
	217
Il est temps de procéder au contrôle de la qualité de ce bouton Annuler ! Utiliser l'état pour implémenter l'annulation Ajout d'annulation aux commandes du ventilateur de plafond Chaque télécommande a besoin d'un mode fête !	220
	221
	222
	225
Utilisation d'une macrocommande Plus d'utilisations du modèle de commande : mise en file d'attente des requêtes Plus d'utilisations du modèle de commande : journalisation des demandes Modèle de commande dans le monde réel	226
	229
	230
Outils pour votre trousse d'outils de conception	231
	233

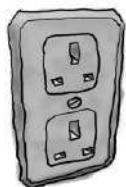
les modèles d'adaptateur et de façade

7 Être adaptatif

Dans ce chapitre, on va essayer des prouesses impossibles, comme mettre une cheville carrée dans un trou rond. Son

impossible? Pas quand on a des modèles de conception. Vous vous souvenez du motif décorateur ? Nous objets enveloppés pour leur confier de nouvelles responsabilités. Nous allons maintenant envelopper certains objets dans un but différent : donner à leurs interfaces l'apparence de quelque chose qu'elles ne sont pas. Pourquoi ferions-nous ça ? Nous pouvons donc adapter une conception attendant une interface à une classe qui implémente une interface différente. Ce n'est pas tout ; pendant qu'on y est, on va regarder un autre modèle qui enveloppe les objets pour simplifier leur interface.

Prise murale britannique



Prise CA standard



Des adaptateurs tout autour de 238

nous Adaptateurs orientés objet 239

S'il marche comme un canard et cancane comme un canard, alors il doit s'agir—
d'une dinde de canard enveloppée avec un adaptateur pour canard... 240

Testez l'adaptateur. 242

243

245

Adaptateurs d'objets et de classes 246

Adaptateurs du monde réel 250

Adaptation d'une énumération à un
itérateur Home Sweet Home Theater 251

Regarder un film (à la dure) 257

Lumières, caméra, façade ! 258

Construire la façade de votre cinéma maison 260

Implémenter l'interface simplifiée Temps pour
regarder un film (en toute simplicité) Modèle
de façade défini 263

Le principe du moindre savoir 264

Comment NE PAS se faire des amis et influencer les objets Le modèle
de façade et le principe du moindre savoir Outils pour votre boîte à
outils de conception 265

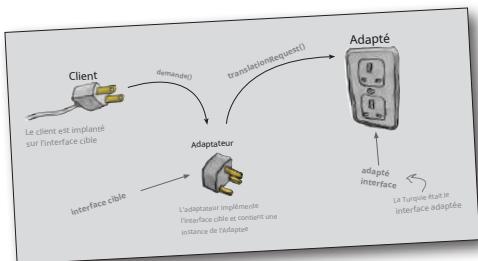
266

267

268

271

272



le modèle de méthode de modèle

8

Algorithmes d'encapsulation

Nous avons encapsulé la création d'objets, l'invocation de méthodes, les interfaces complexes, les canards, les pizzas... quelle pourrait être la prochaine étape ?

On va commencer à encapsuler des morceaux d'algorithmes pour que les sous-classes puissent se connecter directement à un calcul à tout moment. Nous allons même découvrir un principe de conception inspiré d'Hollywood. Commençons...



Il est temps de prendre un peu plus de caféine	278
Préparer des cours de café et de thé (en Java) Faisons	279
abstraction de ce café et de ce thé	282
Pour aller plus loin dans la	283
conception... Extraire prepareRecipe()	284
Qu'avons-nous fait ?	287
Découvrez la méthode modèle	288
Qu'est-ce que la méthode du modèle nous a	290
apporté ? Méthode Modèle Modèle défini	291
Méthode Modèle Accrochée... Utilisation du	294
crochet	295
Le principe hollywoodien et la méthode	299
modèle	301
Trier avec la méthode Template On a	302
quelques canards à trier... Qu'est-ce	303
que compareTo() ?	303
Comparer les canards et les canards	304
Trions quelques canards	305
La fabrication de la machine à trier les canards	306
Swingin' with Frames	308
Listes personnalisées avec les outils AbstractList	309
pour votre boîte à outils de conception	313

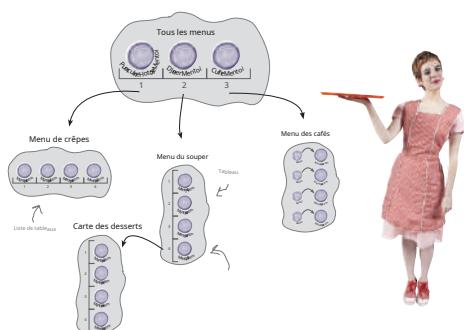
les modèles itérateur et composite

9

Des collections bien gérées

Il y a plusieurs façons d'insérer des objets dans une collection.

Placez-les dans un tableau, une pile, une liste, une carte de hachage : faites votre choix. Chacun a ses propres avantages et compromis. Mais à un moment donné, vos clients voudront parcourir ces objets, et quand ils le feront, allez-vous leur montrer votre implémentation ? On espère certainement que non ! Ce ne serait tout simplement pas professionnel. Eh bien, vous n'êtes pas obligé de risquer votre carrière ; dans ce chapitre, vous allez voir comment permettre à vos clients de parcourir vos objets sans jamais avoir un aperçu de la façon dont vous stockez vos objets. Vous allez aussi apprendre à créer des super collections d'objets capables de parcourir des structures de données impressionnantes en un seul bond. Et si ça ne suffit pas, vous allez aussi apprendre une ou deux choses sur la responsabilité objet.



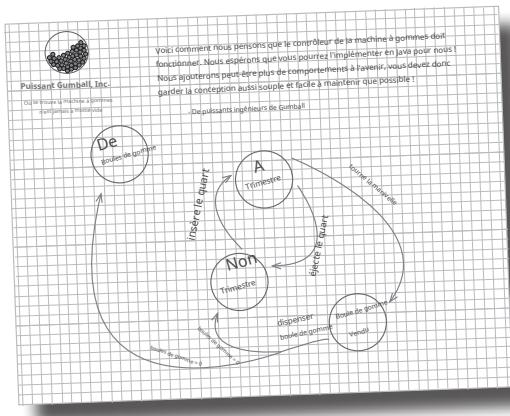
Dernières nouvelles : Objectville Diner et Objectville Pancake House fusionnent	318
Consultez les éléments du menu	319
Mise en œuvre de la spécification : notre première tentative Peut-on encapsuler l'itération ?	323
Découvrez le modèle d'itérateur Ajouter un itérateur à DinerMenu Retravailler le DinerMenu avec Iterator Corriger le code de la serveuse	327
Tester notre code	328
Réviser notre conception actuelle...	330
Nettoyer les choses avec le modèle d'itérateur java.util.Iterator défini	331
La structure du modèle d'itérateur Le principe de responsabilité unique Rencontrer l'interface itérable de Java La boucle for améliorée de Java	333
l'améliorée de Java	335
Consultez les itérateurs et les collections du menu du Café	339
La serveuse est-elle prête pour les heures de grande écoute ? Le modèle composite défini	340
Conception de menus avec composite	343
Implémentation de MenuComponent	344
Implémentation de MenuItem	347
Mise en œuvre du menu composite	355
Maintenant pour le test drive...	360
Outils pour votre trousse d'outils de conception	363
	366
	369
	376

le modèle d'État

10 L'état des choses

Un fait peu connu : les modèles de stratégie et d'état étaient jumeaux séparés à la naissance.

On pourrait penser qu'ils vivraient des vies similaires, mais le modèle de stratégie a continué à créer une entreprise extrêmement prospère autour d'algorithmes interchangeables, tandis que State a choisi la voie peut-être plus noble d'aider les objets à contrôler leur comportement en modifiant leur état interne. Aussi différents que soient leurs chemins, vous trouverez en dessous presque exactement le même design. Comment c'est possible ? Comme vous le verrez, Stratégie et État ont des intentions bien différentes. Commençons par creuser et voir ce qu'est le modèle d'état, puis nous reviendrons pour explorer leur relation à la fin du chapitre.



Disjoncteurs Java	382
Machines à états 101	384
Écrire le code	386
Essais internes	388
Vous saviez que ça allait arriver... une demande de changement ! L'ÉTAT désordonné des choses...	390
Le nouveau design	394
Définir les interfaces et les classes d'état	395
Retravailler la machine à billes de Gumball	398
Maintenant, regardons la classe GumballMachine complète... Mise en œuvre de plus d'états	399
Le modèle d'état défini	406
On doit encore finir le jeu Gumball 1 sur 10.	409
Terminer le jeu	410
Démo pour le PDG de Mighty Gumball, Inc. Contrôle de la santé mentale...	411
On avait presque oublié !	413
Outils pour votre trousse d'outils de conception	416
	419

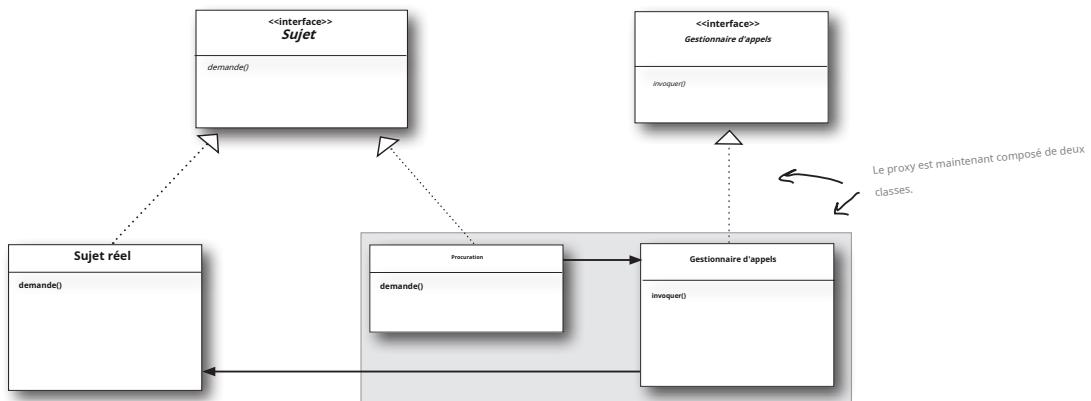


1 1 Contrôle de l'accès aux objets

Avez-vous déjà joué le bon flic et le méchant flic ? Vous êtes le bon flic et vous fournissez tous vos services de manière gentille et amicale, mais vous ne voulez pas que tout le monde vous demande des services, alors vous avez le mauvais flic qui contrôle l'accès à vous. C'est ce que font les proxys : contrôler et gérer les accès. Comme vous le verrez, les proxys remplacent les objets qu'ils mandatent de plusieurs manières. Les proxys sont connus pour acheminer des appels de méthodes entiers sur Internet pour leurs objets mandatés ; ils sont aussi connus pour remplacer patiemment certains articles assez paresseux.



Codage du moniteur	427
Test du moniteur	428
Méthodes à distance 101	433
Préparer la GumballMachine à devenir un service distant	446
Inscription au registre RMI...	448
Le modèle de proxy défini	455
Préparez-vous pour le proxy virtuel	457
Concevoir la couverture de l'album	459
Proxy virtuel	460
Écrire le proxy d'image	469
Utiliser le proxy de l'API Java pour créer un proxy de protection	470
Geeky Matchmaking dans Objectville	471
Mise en œuvre de The Person Drame de cinq minutes : protéger les sujets	473
Vue d'ensemble : créer un proxy dynamique pour la personne	474
The Proxy Zoo	482
Outils pour votre boîte à outils de conception	485
Le code de la visionneuse de pochettes d'album	489

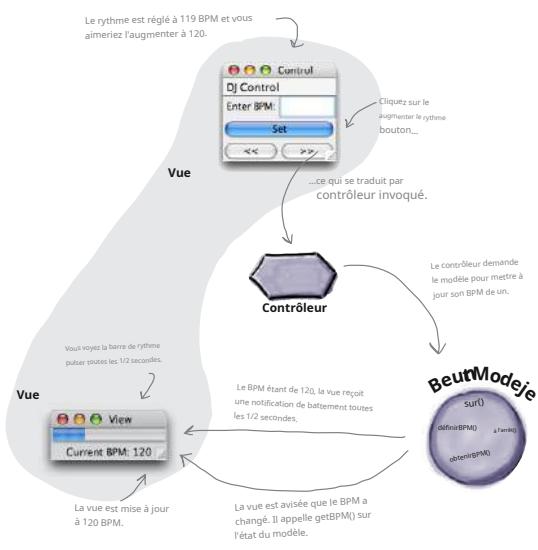


motifs composés

12 Modèles de modèles

Qui aurait cru que les modèles pouvaient fonctionner ensemble?

Vous avez déjà été témoin des discussions acrimonieuses au coin du feu (et vous n'avez même pas vu les pages Pattern Death Match que l'éditeur nous a forcés à supprimer du livre), alors qui aurait pensé que les modèles pouvaient réellement bien s entendre ? Eh bien, croyez-le ou non, certaines des conceptions OO les plus puissantes utilisent plusieurs modèles ensemble. Préparez-vous à faire passer vos compétences en matière de modèles au prochain niveau ; c'est le temps pour les modèles composés.



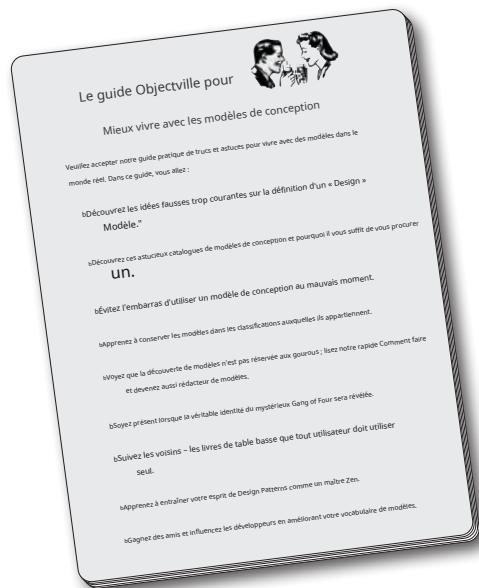
Travailler ensemble	494
Rencontre de canards	495
Qu'avons-nous fait ?	517
Vue plongeante : le diagramme de classes	518
Le roi des modèles composés	520
Rencontrez Model-View-Controller Un	523
examen plus approfondi...	524
Comprendre MVC comme un ensemble de motifs	526
Utiliser MVC pour contrôler le rythme... Construire les pièces	528
Jetons maintenant un coup d'œil à la classe concrète	531
BeatModel The View	532
Mise en œuvre de la vue	533
Passons maintenant au contrôleur.	534
Adaptation du modèle	536
Et maintenant, pour un test... Des outils pour votre boîte à outils de conception	538
	539
	540
	542
	545



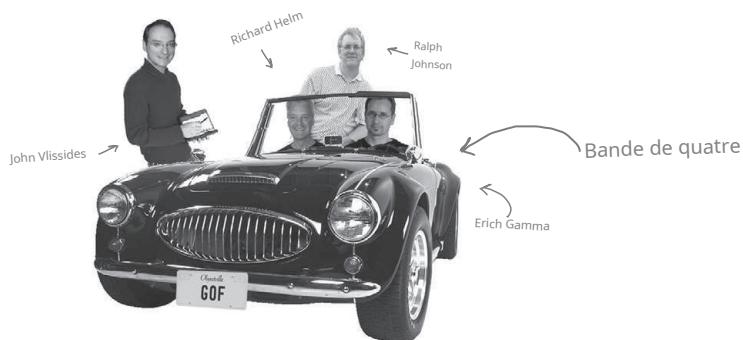
mieux vivre avec des schémas

13 Modèles dans le monde réel

Ahhh, maintenant t'es prêt pour un nouveau monde brillant rempli avec des modèles de conception. Mais avant d'ouvrir toutes ces nouvelles portes d'opportunités, nous devons aborder quelques détails que vous rencontrerez dans le monde réel. Venez, nous avons un joli guide pour vous aider dans la transition...

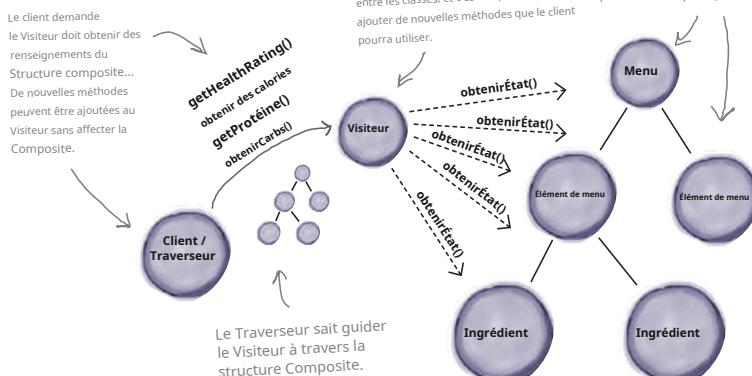


Modèle de conception défini	565
En regardant de plus près la définition du Design Pattern	567
Que la force soit avec vous	568
Alors tu veux devenir un rédacteur de Design Patterns.	573
Penser selon des modèles	575
Votre esprit sur les modèles	580
N'oubliez pas le pouvoir du vocabulaire partagé	583
Cruisin' Objectville avec le Gang of Four Votre voyage ne fait que commencer...	587
Le zoo des motifs	588
Anéantir le mal avec des outils anti-modèles	590
pour votre boîte à outils de conception	592
Quitter Objectville	594
	595



14 Annexe : autres modèles

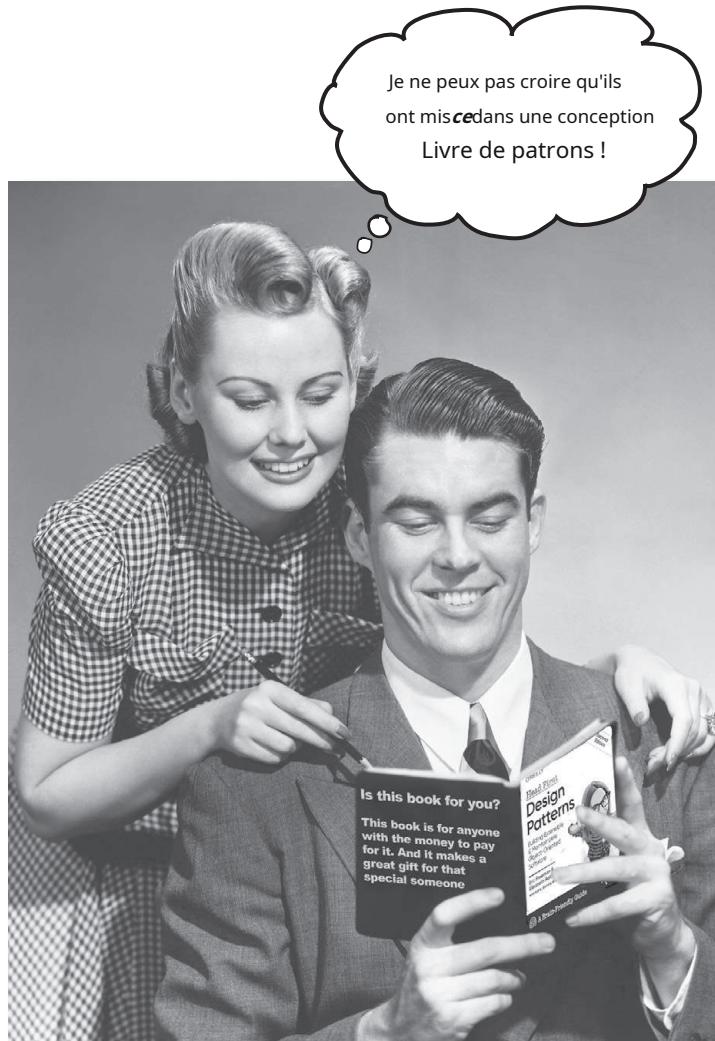
Tout le monde ne peut pas être le plus populaire. Beaucoup de choses ont changé dans le depuis plus de 25 ans. Depuis *Modèles de conception : éléments de logiciels orientés objet réutilisables* est sorti pour la première fois, les développeurs ont appliqué ces modèles des milliers de fois. Les modèles que nous résumons dans cette annexe sont des modèles officiels du GoF à part entière, portant des cartes, mais ne sont pas utilisés aussi souvent que les modèles que nous avons explorés jusqu'à présent. Mais ces modèles sont impressionnantes en eux-mêmes, et si votre situation l'exige, vous devez les appliquer la tête haute. Notre objectif dans cette annexe est de vous donner une idée générale de la nature de ces modèles.



Pont	598
Constructeur	600
Chaîne de responsabilité poids mouche	602
Interprète	604
Médiateur	606
Memento	608
Prototype	610
Visiteur	612
	614

comment utiliser çaivre

Introduction



Dans cette section, on répond à la question brûlante :
« Alors, pourquoi ont-ils mis ça dans un livre de modèles de conception ?

À qui s'adresse ce livre ?

Si vous pouvez répondre « oui » à toutes ces questions :

- ① Vous savez Java (vous n'avez pas besoin d'être un gourou) ou un autre langage orienté objet ?
- ② Voulez-vous apprendre, comprendre, se rappeler, et postuler les modèles de conception, y compris les principes de conception OO sur lesquels sont basés les modèles de conception ?
- ③ Préférez-vous conversation stimulante lors d'un souper des cours académiques secs et ennuyants ?

ce livre est pour vous.

Tous nos exemples sont en Java, mais vous devriez être capable de comprendre les concepts principaux du livre si vous connaissez un autre langage orienté objet.

Qui devrait probablement se retirer de ce livre ?

Si vous pouvez répondre « oui » à l'une de ces questions :

- ① Êtes-vous complètement nouveau dans **programmation orientée objet** ?
- ② Êtes-vous un concepteur/développeur orienté objet à la recherche d'un **référence** livre ?
- ③ Êtes-vous un architecte à la recherche **entreprise** des modèles de conception ?
- ④ Es-tu père d'essayer quelque chose de différent? Préféreriez-vous avoir un canal radiculaire plutôt que de mélanger des rayures avec du plaid? Croyez-vous qu'un livre technique ne peut pas être sérieux si les concepts orientés objet sont anthropomorphisés?

ce livre n'est pas pour vous.



[Remarque du marketing : ce livre s'adresse à toute personne possédant une carte de crédit.]

On sait ce que vous pensez.

« Comment cela peut-il être un livre de programmation sérieux ?

Qu'est-ce que c'est que tous ces graphiques?

« Puis-je vraiment l'apprendre de cette façon ?

Et on sait ce que pense ton cerveau.

Votre cerveau a soif de nouveauté. C'est toujours la recherche, le scan, attendre pour quelque chose d'inhabituel. Il a été construit de cette façon et ça vous aide à rester en vie.

Aujourd'hui, vous êtes moins susceptible d'être une collation de tigre. Mais ton cerveau cherche toujours. On ne sait jamais.

Alors, que fait ton cerveau de toutes les choses routinières, ordinaires et normales que tu rencontres ? Tout ça conserve pour les empêcher d'interférer avec le cerveau réel travail : consigner des choses qui comptent. Ça ne prend pas la peine de sauvegarder les choses ennuyeuses ; ils ne dépassent jamais le filtre « ce n'est évidemment pas important ».

Comment votre cerveau fonctionne-t-il *savoir* qu'est-ce qui est important ? Supposons que vous partiez en randonnée d'une journée et qu'un tigre saute devant vous, qu'est-ce qui se passe dans votre tête et dans votre corps ?

Les neurones se déclenchent. Les émotions montent. *Augmentation des produits chimiques.*

Et c'est comme ça que ton cerveau sait...

Ça doit être important ! N'oubliez pas ça !

Mais imaginez que vous êtes à la maison ou dans une bibliothèque. C'est un endroit sécuritaire, chaud et sans tigres. Vous étudiez. Se préparer pour un examen. Ou essayez d'apprendre un sujet technique difficile qui, selon votre patron, prendra une semaine, dix jours au maximum.

Juste un problème. Ton cerveau essaie de te rendre un grand service. Il essaie de s'assurer que ça *évidemment* contenu sans importance n'encombre pas les ressources rares. Des ressources mieux utilisées pour entreposer les très gros objets. Comme des tigres. Comme le risque d'incendie. Comme si vous ne devriez plus jamais faire de planche à neige en short.

Et il n'y a pas de moyen facile de dire à votre cerveau : « Hé cerveau, merci beaucoup, mais peu importe à quel point ce livre est ennuyeux et à quel point je m'inscris peu sur l'échelle émotionnelle de Richter en ce moment, je veux vraiment que tu continues.



Nous considérons le lecteur « tête première » comme un apprenant.

Alors qu'est-ce qu'il faut pour apprendre quelque chose? Tout d'abord, vous devez obtenir alors assurez-vous de pas le faire oublierça. Il ne s'agit pas de vous mettre des faits dans la tête. Basé sur les dernières recherches en sciences cognitives, en neurobiologie et en psychologie éducative, l'apprentissage prend bien plus que du texte sur une page. On sait ce qui excite ton cerveau.

Certains des principes d'apprentissage Head First :

Rendez-le visuel. Les images sont bien plus mémorisables que les mots seuls et rendent l'apprentissage beaucoup plus efficace (jusqu'à 89 % d'amélioration des études de rappel et de transfert). Ça rend aussi les choses plus compréhensibles. Placez les mots à l'intérieur ou à proximité des graphiques ils se rapportent, plutôt qu'en bas ou sur une autre page, et les apprenants auront jusqu'à deux fois comme susceptible de résoudre les problèmes liés au contenu.



Utiliser un style conversationnel et personnalisé.

Dans les études, les étudiants obtiennent des résultats jusqu'à 40 % supérieurs aux tests post-apprentissage si le contenu s'adresse directement à le lecteur, en utilisant un style conversationnel à la première personne plutôt que d'adopter un style formel tonifiant. Racontez des histoires au lieu de faire la leçon. Utilisez un langage informel. Ne vous prenez pas trop au sérieux. Ce qui accorde plus d'attention à : un compagnon de souper stimulant ou une conférence ?

C'est vraiment poche d'être une méthode abstraite.
Vous n'avez pas de corps.



itinérance vide abstraite () ;

Pas de corps de méthode !
Termez-le par un point-virgule.

Amener l'apprenant à réfléchir plus profondément. Autrement dit, à moins que vous ne fassiez travailler vos neurones activement, il ne se passe pas grand-chose dans votre tête. UN le lecteur doit être motivé, engagé, curieux et inspiré pour résoudre des problèmes, tirer des conclusions et générer de nouvelles connaissances. Et pour ça, il faut des défis, des exercices, des questions qui font réfléchir, activités qui impliquent les deux côtés du cerveau et de multiples sens.

Est-il logique de dire que Tub IS-A Bathroom ? La salle de bain EST-UNE baignoire ? Où est-ce une relation HAS-A ?



Attirez et gardez l'attention du lecteur. Nous avons tous vécu l'expérience « Je veux vraiment apprendre ça, mais je ne peux pas rester éveillé après la première page ». Votre cerveau prête attention aux choses qui sont hors du commun, intéressants, étranges, accrocheurs, inattendus. Apprendre un nouveau sujet technique et difficile n'a pas besoin d'être ennuyeux. Votre cerveau apprendra beaucoup plus vite si ce n'est pas le cas.

Touchez leurs émotions. Nous savons maintenant que votre capacité à vous rappeler de quelque chose dépend en grande partie de son contenu émotionnel. Tu te souviens de ce que tu as insinué propos de. Tu te rappelles quand tu as senti quelque chose. Non, on ne parle pas d'histoires déchirantes sur un garçon et son chien. On parle d'émotions comme la surprise, la curiosité, le plaisir, « qu'est-ce que... ? , et le sentiment de « Je règne ! ça arrive lorsque vous résolvez une énigme, apprenez quelque chose que tout le monde considère comme difficile ou réalisez que vous savez quelque chose que « je suis plus technique que toi » Bob de l'ingénierie ne fait pas.



Métacognition : penser à penser

Si vous voulez vraiment apprendre, et si vous voulez apprendre plus vite et plus profondément, faites attention à la façon dont vous êtes attentif. Pensez à la façon dont vous pensez. Apprenez comment vous apprenez.

La plupart d'entre nous n'ont pas suivi de cours sur la métacognition ou la théorie de l'apprentissage dans notre enfance. On était *attendu* à apprendre, mais rarement *enseigné* à apprendre.

Mais nous présumons que si vous tenez ce livre, vous voulez vraiment apprendre les modèles de conception. Et vous ne voulez probablement pas y consacrer beaucoup de temps. Et tu veux *se rappeler* ce que vous lisez et être capable de l'appliquer. Et pour ça, il faut *comprendre ça*. Pour tirer le maximum de ce livre, *outout* livre ou expérience d'apprentissage, prenez la responsabilité de votre cerveau. Ton cerveau est allumé *ce contenu*.

Le truc, c'est de faire en sorte que votre cerveau considère le nouveau matériel que vous apprenez comme étant vraiment important. Indispensable pour votre bien-être. Aussi important qu'un tigre. Sinon, vous vous retrouverez dans une bataille constante, votre cerveau faisant de son mieux pour empêcher le nouveau contenu de coller.

Alors, comment faire croire à votre cerveau que les motifs de conception sont aussi importants qu'un tigre ?

Il existe une méthode lente et fastidieuse, ou une méthode plus rapide et plus efficace. La voie lente est une pure répétition. Vous savez évidemment que vous êtes capable d'apprendre et de se souvenir même des sujets les plus ennuyeux, si vous continuez martelant sur la même chose. Avec suffisamment de répétitions, votre cerveau dit : « Ça veut pas dire *sentir* important pour lui, mais il continue de regarder la même chose *plus deet plus deet plus de*, donc j'imagine que ça doit être le cas.

Le moyen le plus rapide est de faire ***tout ce qui augmente l'activité cérébrale***, particulièrement différent *genres de l'activité cérébrale*. Les éléments de la page précédente constituent une grande partie de la solution, et il a été prouvé qu'ils aident votre cerveau à travailler en votre faveur. Par exemple, des études montrent que mettre des mots *dans* les images qu'elles décrivent (par opposition à un autre endroit dans la page, comme une légende ou dans le corps du texte) amènent votre cerveau à essayer de comprendre la relation entre les mots et l'image, ce qui provoque le déclenchement d'un plus grand nombre de neurones. Plus de neurones s'activent = plus de chances pour votre cerveau de *obtenir* que c'est quelque chose qui mérite qu'on y prête attention, et éventuellement qu'on l'enregistre.

Un style de conversation est utile parce que les gens ont tendance à être plus attentifs lorsqu'ils perçoivent qu'ils sont dans une conversation, car ils sont censés suivre et tenir leur bout. Ce qui est étonnant, c'est que votre cerveau ne le fait pas nécessairement *soinsque* la « conversation » est entre vous et un livre ! D'un autre côté, si le style d'écriture est formel et sec, votre cerveau le perçoit de la même manière que vous ressentez une conférence alors que vous êtes assis dans une salle remplie de participants passifs. Pas besoin de rester éveillé.

Mais les images et le style de conversation ne sont que le début.



Voici ce que NOUS avons fait :

On a utilisé **des photos**, parce que votre cerveau est réglé pour les visuels, pas pour le texte. Pour votre cerveau, une image vaut vraiment 1 024 mots. Et lorsque le texte et les images fonctionnent ensemble, nous avons intégré le texte dans les images parce que votre cerveau fonctionne plus efficacement lorsque le texte se trouve dans l'objet auquel il fait référence, plutôt que dans une légende ou enfoui ailleurs.

On a utilisé **redondance**, disant la même chose dans *différent* de différentes façons et avec différents types de médias, *et plusieurs sens*, pour augmenter les chances que le contenu soit codé dans plusieurs régions de votre cerveau.

Nous avons utilisé des concepts et des images dans *inattendu* façons parce que votre cerveau est réglé pour la nouveauté, et nous avons utilisé des images et des idées avec au moins certains **émotionnel** contenu, parce que votre cerveau est réglé pour porter attention à la biochimie des émotions. Ce qui te pousse à *sentir* quelque chose est plus susceptible d'être rappelé, même si ce sentiment n'est rien de plus qu'un petit **humour, surprise, ou intérêt**.

Nous avons utilisé un formulaire personnalisé, **style conversationnel**, parce que votre cerveau est réglé pour accorder plus d'attention lorsqu'il croit que vous êtes en conversation que s'il pense que vous écoutez passivement une présentation. Ton cerveau fait ça même quand t'es *lecture*.

Nous avons inclus plus de 90 **activités**, parce que votre cerveau est réglé pour apprendre et mémoriser davantage lorsque *vous fait* des choses que lorsque vous lisez des choses. Et on a rendu les exercices difficiles mais faisables, parce que c'est ce que la plupart *gens* préférer.

On a utilisé **plusieurs styles d'apprentissage**, parce que tu peut préférer les procédures étape par étape, tandis que quelqu'un d'autre veut d'abord comprendre la situation dans son ensemble, tandis que quelqu'un d'autre veut simplement voir un exemple de code. Mais quelle que soit votre préférence d'apprentissage, *tout le monde* bénéficie de voir le même contenu représenté de plusieurs manières.

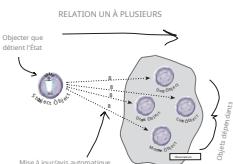
Nous avons inclus du contenu pour *les deux côtés de ton cerveau*, parce que plus votre cerveau est engagé, plus vous avez de chances d'apprendre et de vous souvenir, et plus vous pouvez rester concentré longtemps. Étant donné que travailler un côté du cerveau signifie souvent donner à l'autre côté une chance de se reposer, vous pouvez être plus productif en apprenant sur une période plus longue.

Et nous avons inclus **histoires** et des exercices qui présentent **plus d'un point de vue**, parce que votre cerveau est configuré pour apprendre plus en profondeur lorsqu'il est obligé de faire des évaluations et des jugements.

Nous avons inclus **défis**, à la fois avec des exercices et en demandant **des questions** qui n'ont pas toujours de réponse claire, parce que votre cerveau est réglé pour apprendre et se rappeler quand il le faut *travail* à quelque chose. Pensez-y : vous ne pouvez pas obtenir votre *corps* en forme juste par *regarder* les gens au gymnase. Mais nous avons fait de notre mieux pour nous assurer que lorsque vous travaillez fort, c'est sur le *droit* choses. Ce **vous ne dépensez pas une dent** **répondre** à traiter un exemple difficile à comprendre ou à analyser un texte difficile, chargé de jargon ou trop laconique.

On a utilisé **gens**. Dans des histoires, des exemples, des images, etc., parce que, ben, parce que *tu es* une personne. Et ton cerveau accorde plus d'attention à *gens à choses*.

On a utilisé un **80/20** approche. Nous présumons que si vous envisagez un doctorat en conception de logiciels, ce ne sera pas votre seul livre. Alors on ne parle pas de tout. Juste ce que tu vas vraiment **besoin**.



Le gourou des modèles

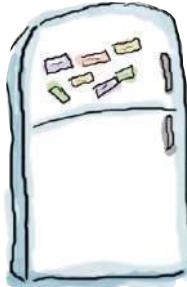


POINTS À PUCE



Des énigmes





Voici ce que VOUS pouvez faire pour plier votre cerveau à la soumission

On a donc fait notre part. Le reste dépend de vous. Ces conseils sont un point de départ ; écoutez votre cerveau et déterminez ce qui fonctionne pour vous et ce qui ne fonctionne pas. Essayez de nouvelles choses.

Coupez-le et collez-le sur votre réfrigérateur.

① Ralentir. Plus vous comprenez, moins vous devez mémoriser.

Ne te contente pas/ire. Arrêtez-vous et réfléchissez. Lorsque le livre vous pose une question, ne répondez pas simplement directement. Imaginez que quelqu'un pose réellement la question. Plus vous forcez votre cerveau à réfléchir profondément, plus vous avez de chances d'apprendre et de vous souvenir.

② Faites les exercices. Écrivez vos propres notes.

On les a mis en place, mais si on les faisait pour vous, ce serait comme si quelqu'un d'autre faisait vos séances d'entraînement à votre place. Et ne fais pas que/heureux exercices. Utilisez un crayon. Il existe de nombreuses preuves que l'activité physique pendant quel'apprentissage peut augmenter l'apprentissage.

③ Lisez « Il n'y a pas de questions niaiseuses »

Ça veut dire tous. Ce ne sont pas des barres latérales facultatives... **ils font partie du contenu principal!** Ne les sautez pas.

④ Faites-en la dernière chose que vous lisez avant de vous coucher. Ou du moins le dernier stimulant chose.

Une partie de l'apprentissage (notamment le transfert vers la mémoire à long terme) se produit après que vous ayez déposé le livre. Votre cerveau a besoin de temps seul pour faire plus de traitements. Si vous ajoutez quelque chose de nouveau pendant ce temps de traitement, une partie de ce que vous venez d'apprendre sera perdue.

⑤ Buvez de l'eau. Beaucoup.

Votre cerveau fonctionne mieux dans un bon bain de liquide. La déshydratation (qui peut survenir avant même d'avoir soif) diminue la fonction cognitive.

⑥ Parlez-en. À voix haute.

Parler active une autre partie du cerveau. Si vous essayez de comprendre quelque chose ou si vous augmentez vos chances de vous en souvenir plus tard, dites-le à voix haute. Mieux encore, essayez de l'expliquer à voix haute à quelqu'un d'autre. Vous apprendrez plus vite et vous découvrirez peut-être des idées dont vous ignoriez l'existence lorsque vous lisiez à ce sujet.

⑦ Écoutez votre cerveau.

Faites attention à savoir si votre cerveau est surchargé. Si vous commencez à survoler la surface ou à oublier ce que vous venez de lire, il est temps de prendre une pause. Une fois que vous avez dépassé un certain point, vous n'apprendrez pas plus vite en essayant d'en mettre plus, et vous pourriez même nuire au processus.

⑧ Sentir quelque chose!

Votre cerveau a besoin de savoir que ceci importe. Impliquez-vous dans les histoires. Créez vos propres légendes pour les photos. Gémir après une mauvaise blague, c'est encore mieux que de ne rien ressentir du tout.

⑨ Concevoir quelque chose!

Appliquez ceci à quelque chose de nouveau que vous concevez ou refactorisez un projet plus ancien. Fais juste quelque chose pour acquérir de l'expérience au-delà des exercices et des activités de ce livre. Tout ce dont vous avez besoin est un crayon et un problème à résoudre... un problème qui pourrait bénéficier d'un ou plusieurs modèles de conception.

Lisez-moi

Il s'agit d'une expérience d'apprentissage, pas d'un ouvrage de référence. Nous avons délibérément enlevé tout ce qui pourrait nuire à l'apprentissage du sujet sur lequel nous travaillons à ce stade du livre. Et la première fois, vous devez commencer par le début, car le livre fait des hypothèses sur ce que vous avez déjà vu et appris.

Nous utilisons une version
plus simple et modifiée
d'UML.


On utilise des diagrammes simples de type UML.

Bien qu'il y ait de fortes chances que vous ayez déjà rencontré UML, ce sujet n'est pas abordé dans le livre et ce n'est pas un prérequis pour le livre. Si vous n'avez jamais vu UML auparavant, ne vous inquiétez pas, on va vous donner quelques conseils en cours de route. En d'autres mots, vous n'aurez pas à vous soucier simultanément des Design Patterns et d'UML. Nos diagrammes sont « *UML-semblable* » – alors qu'on essaie d'être fidèles à UML, il arrive parfois qu'on contourne un peu les règles, généralement pour nos propres raisons artistiques égoïstes.

Directeur
obtenir des films
obtenir les Oscars()
getKevinBaconDegrés()

On ne couvre pas tous les modèles de conception jamais créés.

Il y a un *beaucoup* de modèles de conception : les modèles fondamentaux originaux (appelés modèles GoF), les modèles Java d'entreprise, les modèles architecturaux, les modèles de conception de jeux et bien plus encore. Mais notre but était de s'assurer que le livre pesait moins que la personne qui le lisait, donc on ne les couvre pas tous ici. Nous nous concentrerons sur les modèles fondamentaux qui *questionnent* à partir des modèles originaux orientés objet du GoF, et en vous assurant que vous comprenez vraiment, vraiment et profondément comment et quand les utiliser. Vous trouverez un bref aperçu de certains des autres modèles (ceux que vous êtes beaucoup moins susceptibles d'utiliser) en annexe. Quoi qu'il en soit, une fois que vous en aurez fini avec *Modèles de conception tête première*, vous pourrez récupérer n'importe quel catalogue de modèles et vous mettre rapidement au courant.

Les activités ne sont PAS facultatives.

Les exercices et les activités ne sont pas des compléments ; ils font partie du contenu principal du livre. Certains d'entre eux visent à aider à la mémoire, d'autres à comprendre et d'autres encore à vous aider à appliquer ce que vous avez appris. ***Ne sautez pas les exercices.*** Les mots croisés sont les seules choses que vous ne faites pas *avons* à faire, mais ils sont bons pour donner à votre cerveau une chance de réfléchir aux mots dans un contexte différent.

Nous utilisons le mot « composition » dans le sens général OO, qui est plus souple que l'utilisation stricte du terme « composition » en UML.

Quand on dit « un objet est composé avec un autre objet », on veut dire qu'ils sont liés par une relation HAS-A. Notre utilisation reflète l'utilisation traditionnelle du terme et est celle utilisée dans le texte du GoF (vous apprendrez ce que c'est plus tard). Plus récemment, UML a peu à peu fini ce terme en plusieurs types de composition. Si vous êtes un expert UML, vous pourrez toujours lire le livre et vous devriez pouvoir facilement mapper l'utilisation de la composition à des termes plus raffinés au fur et à mesure de votre lecture.

La redondance est intentionnelle et importante.

Une différence distincte dans un livre Head First est qu'on veut que vous *vraiment* l'obtenir. Et on veut que vous finissiez le livre en vous souvenant de ce que vous avez appris. La plupart des ouvrages de référence ne visent pas la rétention et le rappel, mais ce livre traite de *apprentissage*, vous verrez donc certains des mêmes concepts revenir plus d'une fois.

Les exemples de code sont aussi simples que possible.

Nos lecteurs nous disent qu'il est frustrant de parcourir 200 lignes de code à la recherche des deux lignes qu'ils doivent comprendre. La plupart des exemples de ce livre sont présentés dans le contexte le plus restreint possible, afin que la partie que vous essayez d'apprendre soit claire et simple. Ne vous attendez pas à ce que tout le code soit robuste, ni même complet : les exemples sont écrits spécifiquement pour l'apprentissage et ne sont pas toujours entièrement fonctionnels.

Dans certains cas, nous n'avons pas inclus toutes les instructions d'importation nécessaires, mais nous supposons que si vous êtes un programmeur Java, vous savez que `Liste` de `Tableaux` est dans `java.util`, par exemple. Si les importations ne faisaient pas partie de l'API JSE de base normale, nous le mentionnons. Nous avons aussi placé tout le code source sur le Web pour que vous puissiez le télécharger. Vous le trouverez à <http://wickedlysmart.com/head-first-design-patterns>.

De plus, afin de nous concentrer sur l'apprentissage du code, nous n'avons pas placé nos classes dans des paquets (en d'autres termes, elles sont toutes dans le paquet Java par défaut). Nous ne le recommandons pas dans le monde réel, et lorsque vous téléchargez les exemples de code de ce livre, vous constaterez que toutes les classes *étées* en paquets.

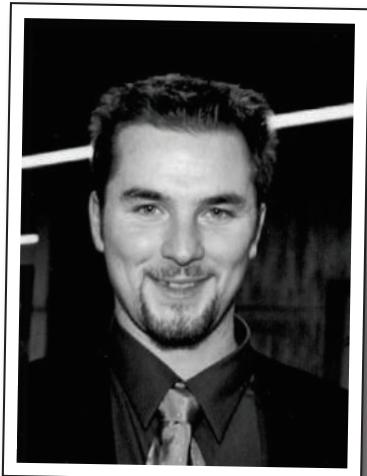
Les exercices Brain Power n'ont pas de réponses.

Pour certains d'entre eux, il n'y a pas de bonne réponse, et pour d'autres, une partie de l'expérience d'apprentissage des activités Brain Power consiste à décider si et quand vos réponses sont bonnes. Dans certains exercices Brain Power, vous trouverez des trucs pour vous orienter dans la bonne direction.

Valentin Crétaz

Réviseurs techniques

Jeff Cums



Ike Van Atta ↗



Johannes de Jong ↗

Leader intrépide de
l'extrême HFDP
Équipe de révision

Jason Ménard



À la mémoire de Philippe Maquet, 1960 - 2004. Votre incroyable expertise technique, votre enthousiasme incessant et votre profonde préoccupation pour l'apprenant nous inspireront toujours.



Philippe Maquet



Dirk Schreckmann



Marc Spritzler ↙

Réviseurs techniques, 2e édition



Julien Setiawan



Georges Heineman



David Pouvoirs



Trisha Gee

↑ MVP des critiques de la 2e édition !

Remerciements

Dès la première édition

Chez O'Reilly :

Notre plus grand merci à **Mike Loukides** chez O'Reilly, pour avoir tout lancé et contribué à façonner le concept Head First en une série. Et un gros merci à la force motrice derrière Head First, **Tim O'Reilly**. Merci à l'astucieux « série maman » Head First **Kyle Hart**, « Dans le roi du design » **Ron Bilodeau**, rock and roll star **Ellie Volkhausen** pour elle inspirée conception de couverture **Mélanie Yarbrough** pour la production pastorale, **Colleen Gorman** et **Rachel Monaghan** pour leurs retouches hardcore, et **Bob Pfahler** pour un indice bien amélioré. Enfin, grâce à **Mike Hendrickson** et **Meghan Blanchette** pour avoir défendu ce livre et bâti l'équipe.

Nos critiques intrépides :

Nous sommes extrêmement reconnaissants envers notre directeur de la revue technique **Johannes de Jong**. Tu es notre héros, Johannes. Et nous apprécions profondément les contributions du cogestionnaire du **Javaranch** l'équipe d'examen, le regretté **Philippe Maquet**. À vous seul, vous avez égayé la vie de milliers de développeurs, et l'impact que vous avez eu sur leur (et sur la nôtre) est éternel. **Jeff Cusse** est terriblement doué pour trouver des problèmes dans nos projets de chapitres, et a encore une fois fait une énorme différence pour le livre. Merci Jef ! **Valentin Crétazz** (AOP Guy), qui est avec nous depuis le tout premier livre Head First, a prouvé (comme toujours) à quel point nous avons vraiment besoin de son expertise technique et de sa perspicacité. Vous faites vibrer Valentin (mais vous perdez l'égalité).

Deux nouveaux membres de l'équipe d'examen HF, **Barney Marispini** et **Ike Van Atta**, a fait un travail remarquable sur le livre – vous nous en avez donné vraiment retour crucial. Merci de nous joindre à l'équipe.

Nous avons également reçu une excellente aide technique de la part des modérateurs/gourous de Javaranch. **Marc Spritzler**, **Jason Ménard**, **Dirk Schreckmann**, **Thomas-Paul**, et **Margarita Isaïeva**. Et comme toujours, grâce notamment au Trail Boss de javaranch.com, **Paul Wheaton**.

Merci aux finalistes du Javaranch « Choisissez Modèles de conception tête première Concours "Couverture". La gagnante, Si Brewster, a soumis l'essai gagnant qui nous a convaincus de choisir la femme que vous voyez sur notre couverture. Les autres finalistes incluent Andrew Esse, Gian Franco Casula, Helen Crosbie, Pho Tek, Helen Thomas, Sateesh Kommineni et Jeff Fisher.

Pour la mise à jour 2014 du livre, nous sommes très reconnaissants envers les réviseurs techniques suivants : George Hoffer, Ted Hill, Todd Bartoszkiewicz, Sylvain Tenier, Scott Davidson, Kevin Ryan, Rich Ward, Mark Francis Jaeger, Mark Masse, Glenn Ray, Bayard.

Remerciements

De la deuxième édition

Chez O'Reilly :

D'abord et avant tout, **Marie Treseler** est la superpuissance qui fait tout arriver et nous lui sommes éternellement reconnaissants pour tout ce qu'elle fait pour O'Reilly, Head First et les auteurs. **Melissa Duffield** et **Michèle Cronin** a dégagé de nombreux chemins qui ont permis la réalisation de cette deuxième édition. **Rachel Monaghana** fait une retouche superbe, donnant un nouvel éclat à notre texte. **Kristen Bruna** rendu le livre magnifique en ligne et sur papier. **Ellie Volckhausen** a fait preuve de magie et a conçu une nouvelle couverture brillante pour la deuxième édition. Merci à tous !

Nos critiques de la 2e édition :

Nous remercions nos réviseurs techniques de la 2e édition d'avoir repris la tâche 15 ans plus tard. **David Pouvoirs** est notre critique de référence (il est à nous, ne pensez même pas à lui demander de réviser *ton* livre) parce qu'il ne manque de rien. **Georges Heineman** est allé au-delà de ses attentes avec ses commentaires, suggestions et rétroactions détaillés, et il a reçu le prix technique MVP de cette édition. **Trisha Gee** et **Julien Setiawan** fourni les connaissances inestimables de Java dont nous avions besoin pour nous aider à éviter ces erreurs Java embarrassantes et dignes de grincer des dents. Merci à tous !

Merci bien spécial

Merci beaucoup à **Erich Gamma**, qui est allé bien au-delà de son devoir en révisant ce livre (il a même emporté un brouillon avec lui en vacances). Erich, votre intérêt pour ce livre nous a inspiré et votre examen technique approfondi l'a considérablement amélioré. Merci également à tous **Bande de quatre** pour leur soutien et leur intérêt, et pour leur apparition spéciale à Objectville. Nous sommes également redéposables à **Quartier Cunningham** et la communauté des modèles qui a créé le Portland Pattern Repository, une ressource indispensable pour nous dans l'écriture de ce livre.

Merci beaucoup à **Mike Loukidès**, **Mike Hendrickson**, et **Meghan Blanchette**. Mike L. était avec nous à chaque étape du processus. Mike, vos commentaires perspicaces ont contribué à façonner le livre et vos encouragements nous ont permis d'avancer. Mike H., merci pour votre persévérance pendant cinq ans à essayer de nous amener à écrire un livre de modèles ; on l'a finalement fait et on est contents d'avoir attendu Head First.

Il faut un village pour écrire un livre technique : **Bill Pugh** et **Ken Arnold** nous a donné des conseils d'experts sur Singleton. **Josué Marinacci** fourni des trucs et des conseils rockin' Swing. **John Brewer** "Pourquoi un canard ? le papier a inspiré SimUDuck (et nous sommes contents qu'il aime aussi les canards). **Dan Friedman** a inspiré l'exemple de Little Singleton. **Daniel Steinberg** a agi comme notre « lien technique » et notre réseau de soutien émotionnel. Merci à Apple **James Dempsey** pour nous avoir permis d'utiliser sa chanson MVC. Et merci à **Richard Warburton**, qui s'est assuré que nos mises à jour du code Java 8 étaient à la hauteur de cette édition mise à jour du livre.

Enfin, un merci personnel au **Équipe d'examen de Javaranch** pour leurs excellentes critiques et leur soutien chaleureux. Vous êtes plus nombreux dans ce livre que vous ne le pensez.

Écrire un livre Head First est une folle aventure avec deux guides touristiques extraordinaires : **Kathy Sierra** et **Bert Bates**. Avec Kathy et Bert, vous rejetez toutes les conventions d'écriture de livres et entrez dans un monde rempli de narration, de théorie de l'apprentissage, de sciences cognitives et de culture pop, où le lecteur règne toujours.

Bienvenue à *

* Modèles de conception



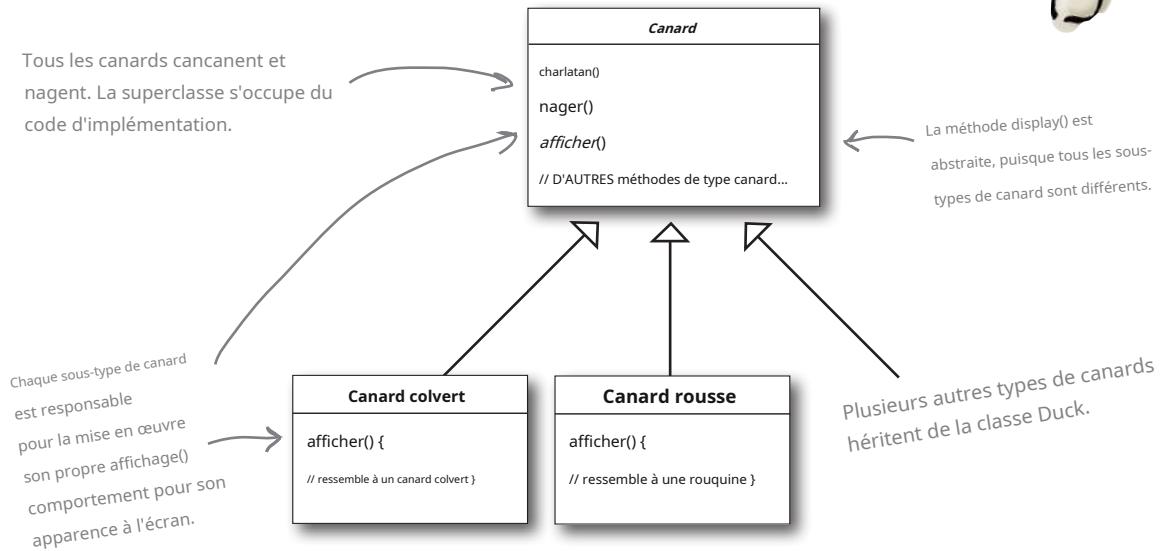
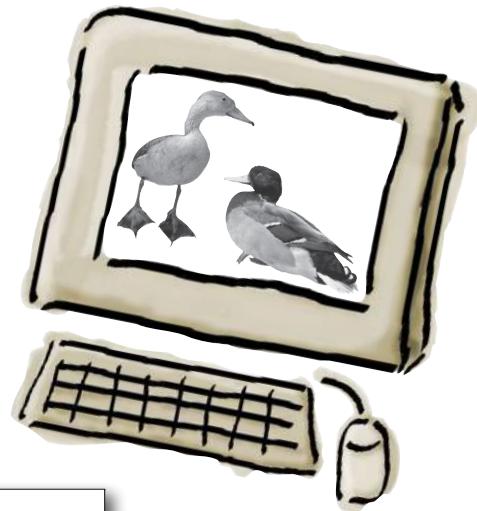
Maintenant qu'on vit
à Objectville, il faut juste
entrez dans les modèles de
conception... nous les faisons. Bientôt
nous serons du mercredi de Jim et Betty
groupe de modèles !

tout le monde
le coup
ouais la nuit

Quelqu'un a déjà réglé vos problèmes. Dans ce chapitre, vous apprendrez pourquoi (et comment) vous pouvez exploiter la sagesse et les leçons apprises par d'autres développeurs qui ont suivi le même chemin de problèmes de conception et ont survécu au voyage. Avant d'avoir terminé, nous examinerons l'utilisation et les avantages des modèles de conception, examinerons quelques principes clés de conception orientée objet (OO) et passerons en revue un exemple du fonctionnement d'un modèle. La meilleure façon d'utiliser les modèles est de *charge ton cerveau avec eux et puis reconnaître des endroits* dans vos conceptions et applications existantes où vous pouvez *appliquez-les*. Au lieu de *coder* à utiliser, avec les modèles que vous obtenez *expériencer* utilisation.

Tout a commencé avec une simple application SimUDuck

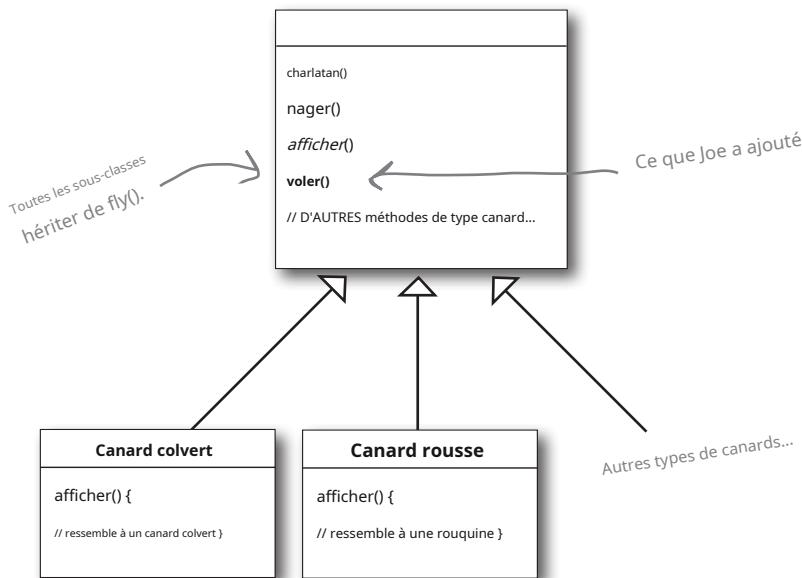
Joe travaille pour une entreprise qui crée un jeu de simulation d'étang aux canards très réussi, *SimUDuck*. Le jeu peut montrer une grande variété d'espèces de canards nageant et émettant des cancans. Les premiers concepteurs du système ont utilisé des techniques OO standard et ont créé une superclasse Duck dont héritent tous les autres types de canards.



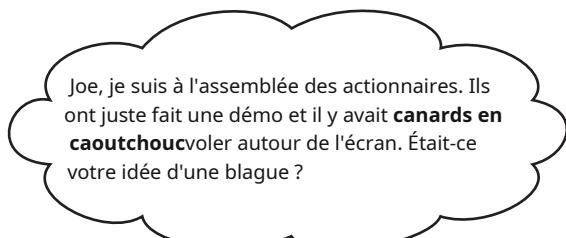
L'an dernier, l'entreprise a été soumise à une pression croissante de la part de ses concurrents. Après une semaine de réflexion hors site sur le golf, les dirigeants de l'entreprise pensent qu'il est temps de faire une grande innovation. Ils ont besoin de quelque chose vraiment impressionnant à montrer lors de la prochaine assemblée des actionnaires à Maui la semaine prochaine.

Mais maintenant on a besoin que les canards VOIENT

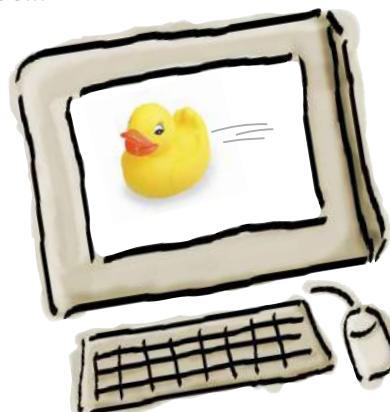
Les dirigeants ont décidé que les canards volants étaient exactement ce dont le simulateur avait besoin pour épater les concurrents. Et bien sûr, le gérant de Joe leur a dit que ce ne serait pas un problème pour Joe de préparer quelque chose dans une semaine. « Après tout », a dit le patron de Joe, « c'est un programmeur OO... à quel point ça peut être difficile ? »



Mais quelque chose s'est terriblement mal passé...



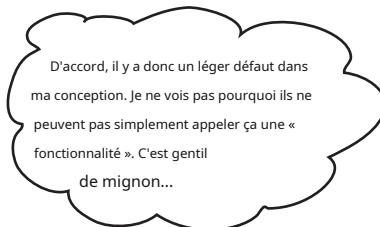
Joe, je suis à l'assemblée des actionnaires. Ils ont juste fait une démo et il y avait **canards en caoutchouc** voler autour de l'écran. Était-ce votre idée d'une blague ?



Que s'est-il passé?

Joe n'a pas remarqué que tous les sous-classes de Duck devraient *voler*. Lorsque Joe a ajouté un nouveau comportement à la superclasse Duck, il a aussi ajouté un comportement qui était *pas approprié* pour certaines sous-classes de Duck. Il a maintenant des objets volants inanimés dans le programme SimUDuck.

Une mise à jour localisée du code a causé un effet secondaire non local (volants en caoutchouc en caoutchouc) !

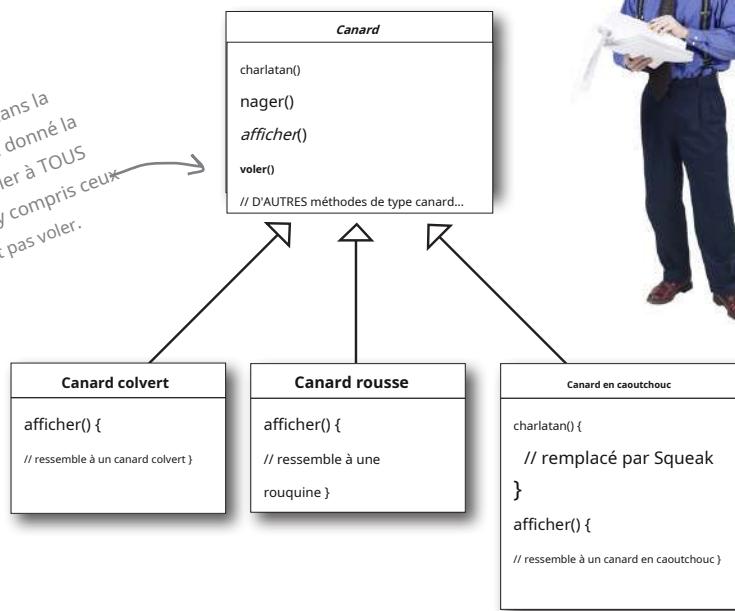


D'accord, il y a donc un léger défaut dans ma conception. Je ne vois pas pourquoi ils ne peuvent pas simplement appeler ça une « fonctionnalité ». C'est gentil de mignon...

Ce que Joe pensait

c'était d'une grande utilité
d'héritage dans le but de réutilisation n'a pas s'est si bien déroulé en matière de maintenance.

En mettant *fly()* dans la superclasse, il a donné la capacité de voler à **TOUT** les canards, y compris ceux ça ne devrait pas voler.



Notez aussi que les canards en caoutchouc ne cancanent pas, donc `quack()` est remplacé par « Squeak ».

Joe pense à l'héritage...



Voici une autre classe dans la hiérarchie ; remarquez que comme RubberDuck, il ne vole pas, mais il ne jase pas non plus.



Sharpen your pencil

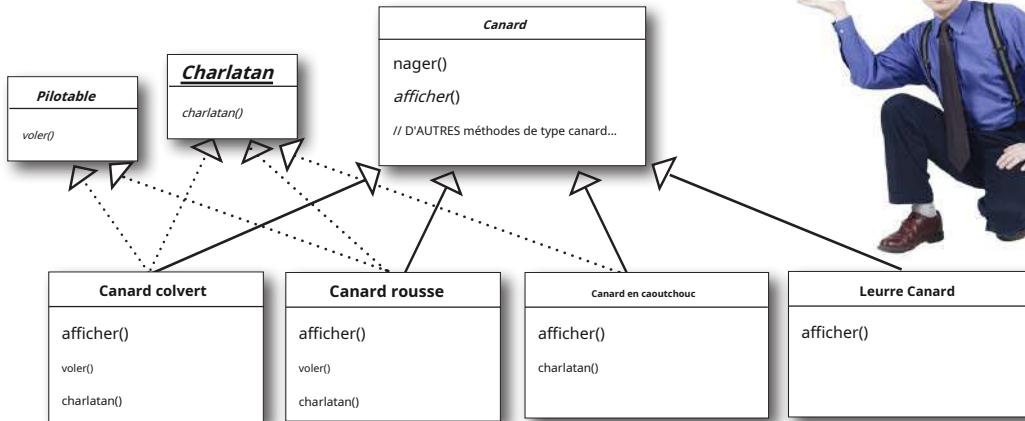
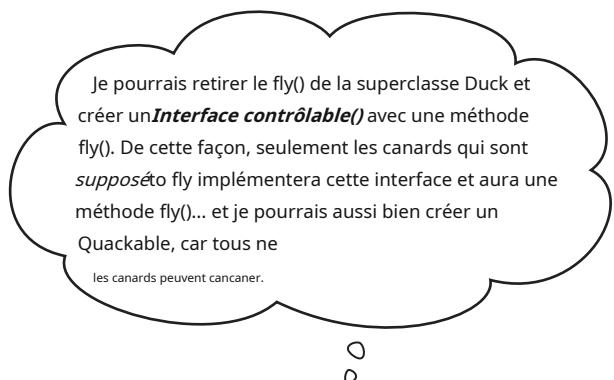
Parmi les éléments suivants, lesquels sont des inconvénients liés à l'utilisation *héritage* pour fournir un comportement de canard ? (Choisissez toutes les réponses qui s'appliquent.)

- A. Le code est dupliqué dans les sous-classes.
- B. Les changements de comportement à l'exécution sont difficiles.
- C. On ne peut pas faire danser les canards.
- D. Il est difficile de connaître tous les comportements des canards.
- E. Les canards ne peuvent pas voler et cancaner en même temps.
- F. Les changements peuvent affecter involontairement d'autres canards.

Comment à propos d'une interface ?

Joe s'est rendu compte que l'héritage n'était probablement pas la solution, car il vient de recevoir un mémo indiquant que les dirigeants veulent désormais mettre à jour le produit tous les six mois (d'une manière qu'ils n'ont pas encore décidé). Joe sait que les spécifications continueront de changer et qu'il sera obligé d'examiner et éventuellement de remplacer `fly()` et `quack()` pour chaque nouvelle sous-classe Duck ajoutée au programme...*pour toujours*.

Il a donc besoin d'un moyen plus propre pour n'avoir que *certain(s)* (mais pas *tous*) des types de canards volent ou cancent.



Que pensez-vous de cette conception ?

C'est l'idée la plus niaiseuse que vous ayez eue.
Pouvez-vous dire « code en double » ? Si tu pensais devoir le faire remplacer quelques méthodes C'était mauvais, comment vas-tu te sentir quand tu devras faire un petit changement à ton comportement de vol... *au total 48 des sous-classes de canard volant ?!*



Quel serait *tus* t'étais Joe ?

On sait que non *tous* des sous-classes devraient avoir un comportement de vol ou de charlatan, donc l'héritage n'est pas la bonne réponse. Mais tout en demandant aux sous-classes d'implémenter des solutions Flyable et/ou Quackable *partie du* problème (pas de canards en caoutchouc volant de manière inappropriée), cela détruit complètement la réutilisation du code pour ces comportements, donc cela crée simplement un *différent* cauchemar d'entretien. Et bien sûr, il peut y avoir plus d'un type de comportement de vol, même parmi les canards qui *fait* voler...

À ce stade, vous attendez peut-être qu'un modèle de conception vienne monter sur un cheval blanc et sauve la situation. Mais à quel point ça serait amusant ? Non, on va trouver une solution à l'ancienne... *en appliquant de bons principes de conception de logiciels OO.*



Ne serait-ce pas un rêve s'il y avait un moyen de créer un logiciel de telle sorte que lorsque nous devons le modifier, nous puissions le faire avec le moins d'impact possible sur le code existant ? On pourrait passer moins de temps à retravailler le code et plus à rendre le programme plus cool choses...

La seule constante en développement de logiciels

D'accord, quelle est la seule chose sur laquelle vous pouvez toujours compter dans le développement de logiciels ?

Peu importe où vous travaillez, ce que vous construisez ou le langage dans lequel vous programmez, quelle est la véritable constante qui vous accompagnera toujours ?

E GNOMIC

(utilisez un miroir pour voir la réponse)

Quelle que soit la façon dont vous concevez une application, au fil du temps, une application doit grandir et changer, sinon elle le fera.*mourir*.



Sharpen your pencil

Beaucoup de choses peuvent mener au changement. Énumérez quelques raisons pour lesquelles vous avez dû modifier le code de vos applications (nous en avons ajouté quelques-unes pour vous aider à démarrer). Vérifiez vos réponses avec la solution à la fin du chapitre avant de continuer.

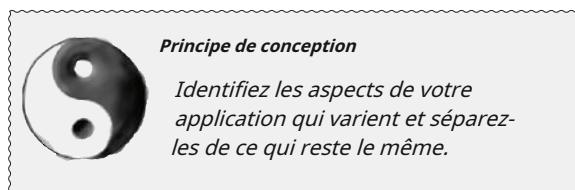
Mes clients ou utilisateurs décident qu'ils veulent autre chose ou qu'ils veulent de nouvelles fonctionnalités.

Mon entreprise a décidé de faire appel à un autre fournisseur de bases de données et achète aussi ses données d'un autre fournisseur qui utilise un format de données différent. Argh !

Se concentrer sur le problème...

Nous savons donc que l'utilisation de l'héritage n'a pas très bien fonctionné, puisque le comportement du canard ne cesse de changer à travers les sous-classes, et ce n'est pas approprié pour *tous*sous-classes pour avoir ces comportements. Les interfaces Flyable et Quackable semblaient prometteuses au début (seuls les canards qui volent réellement seront Flyable, etc.), sauf que les interfaces Java n'ont généralement pas de code d'implémentation, donc pas de réutilisation de code. Dans les deux cas, chaque fois que vous devez modifier un comportement, vous êtes souvent obligé de le retrouver et de le modifier dans toutes les différentes sous-classes où ce comportement est défini, introduisant probablement *nouvelles* bogues en cours de route !

Heureusement, il existe un principe de conception adapté à cette situation.



Le premier de nombreux principes de conception. On va y consacrer plus de temps tout au long du livre.

Autrement dit, si un aspect de votre code change, par exemple à chaque nouvelle exigence, alors vous savez que vous avez un comportement qui doit être retiré et séparé de tout ce qui ne change pas.

Voici une autre façon de penser à ce principe :***prenez les parties qui varient et encapsulez-les, afin de pouvoir plus tard modifier ou étendre les parties qui varient sans affecter celles qui ne varient pas.***

Aussi simple que soit ce concept, il constitue la base de presque tous les modèles de conception. Tous les modèles offrent un moyen de laisser *certaines parties d'un système varient indépendamment de toutes les autres parties*.

D'accord, il est temps de retirer le comportement du canard des cours de canard !

Prenez ce qui varie et « encapsulez-le » pour que ça n'affecte pas le reste de votre code.

Quel est le résultat ? Moins de conséquences involontaires des changements de code et plus de flexibilité dans vos systèmes !

Séparer ce qui change de ce qui reste le même

Par où commencer ? Pour autant qu'on puisse en juger, mis à part les problèmes avec `fly()` et `quack()`, la classe `Duck` fonctionne bien et aucune autre partie de celle-ci ne semble varier ou changer fréquemment. Donc, à part quelques petits changements, on va pratiquement laisser la classe `Duck` tranquille.

Maintenant, pour séparer les « parties qui changent de celles qui restent les mêmes », on va créer deux *ensembles de classes* (complètement en dehors de `Duck`), une pour *voler* et un pour *charlatan*. Chaque ensemble de classes contiendra toutes les implémentations du comportement respectif. Par exemple, on pourrait avoir *un canard qui met en œuvre cancaner*, *un autre qui met en œuvre grinçant*, et *un autre qui met en œuvre silence*.

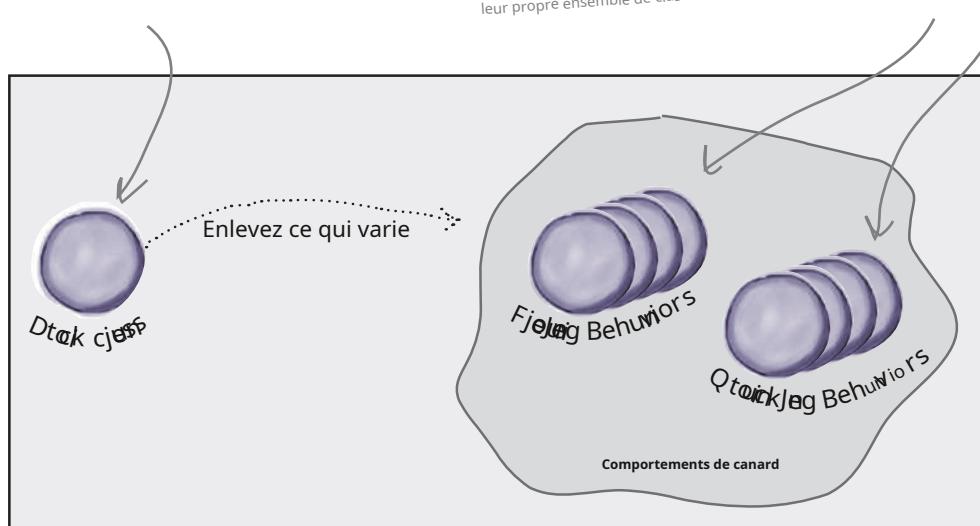
Nous savons que `fly()` et `quack()` sont les parties de la classe `Duck` qui varient selon les canards.

À Séparons ces comportements de la classe `Duck`, nous extrairons les deux méthodes *hors de la classe Duck et créez un nouvel ensemble de classes pour représenter chaque comportement.*

La classe `Canard` est toujours la superclasse de tous les canards, mais on enlève les comportements de mouche et de charlatan et on les place dans une autre structure de classe.

Maintenant, le vol et le cancan ont chacun leur propre ensemble de classes.

Comportements divers les implémentations sont je vais vivre ici.

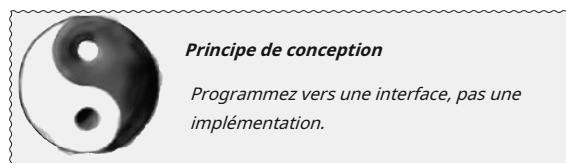


Concevoir les comportements du canard

Alors comment allons-nous concevoir l'ensemble des classes qui mettent en œuvre les comportements de vol et de charlatan ?

On aimerait garder les choses flexibles ; après tout, c'est la rigidité du comportement des canards qui nous a causé des problèmes en premier lieu. Et on sait qu'on veut attribuer des comportements aux instances de Duck. Par exemple, on pourrait vouloir instancier une nouvelle instance de MallardDuck et l'initialiser avec un type de comportement de vol. Et pendant qu'on y est, pourquoi ne pas s'assurer qu'on peut modifier le comportement d'un canard de manière dynamique ? Autrement dit, on devrait inclure des méthodes de définition du comportement dans les classes Duck afin de pouvoir changer le comportement de vol du canard *à tout moment de l'exécution*.

Compte tenu de ces objectifs, examinons notre deuxième principe de conception :



Nous utiliserons une interface pour représenter chaque comportement (par exemple, FlyBehavior et QuackBehavior) et chaque implémentation d'un comportement implémentera l'une de ces interfaces.

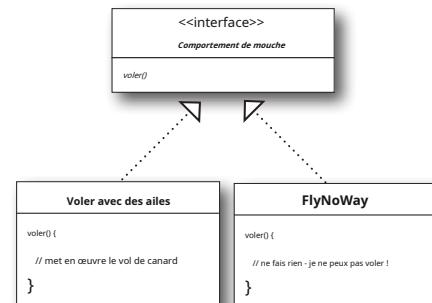
Donc cette fois ce ne sera pas le *Canard* qui implémentera les interfaces de vol et de charlatan. Au lieu de cela, nous créerons un ensemble de classes dont la seule raison de vivre est de représenter un comportement (par exemple, « grincer »), et c'est la *comportement* classe, plutôt que la classe Duck, qui implémentera l'interface de comportement.

Cela contraste avec la façon dont nous faisions les choses auparavant, où un comportement provenait soit d'une implémentation concrète dans la superclasse Duck, soit de la fourniture d'une implémentation spécialisée dans la sous-classe elle-même. Dans les deux cas, on s'appuyait sur une *mise en œuvre*. Nous étions obligés d'utiliser cette implémentation spécifique et il n'y avait aucune possibilité de modifier le comportement (autre que d'écrire plus de code).

Avec notre nouveau design, les sous-classes Duck utiliseront un comportement représenté par une interface (FlyBehavior et QuackBehavior), afin que le réel *mise en œuvre* du comportement (autrement dit, le comportement concret spécifique codé dans la classe qui implémente FlyBehavior ou QuackBehavior) ne sera pas verrouillé dans la sous-classe Duck.

Maintenant, les comportements de Canard vivront dans une classe distincte : une classe qui met en œuvre un comportement particulier interface.

De cette façon, les classes Duck n'auront pas besoin pour connaître les détails de la mise en œuvre pour leurs propres comportements.





Programme à un'interface» signifie en réalité « Programme pour un'supertype.»

Le mot *interface* est surchargé ici. Il y a le *concept* d'une interface, mais il y a aussi le *Java construction* d'une interface. Tu peux *programme vers une interface* sans avoir à utiliser une interface Java. Le but est d'exploiter le polymorphisme en programmant sur un supertype afin que l'objet d'exécution réel ne soit pas verrouillé dans le code. Et nous pourrions reformuler « programme vers un supertype » comme « le type déclaré des variables doit être un supertype, généralement une classe ou une interface abstraite, afin que les objets affectés à ces variables puissent être de n'importe quelle implémentation concrète du supertype, ce qui signifie la classe qui les déclare n'a pas besoin de connaître les types d'objets réels !

C'est probablement une vieille nouvelle pour vous, mais juste pour être sûr qu'on dit tous la même chose, voici un exemple simple d'utilisation d'un type polymorphe : imaginez une classe abstraite Animal, avec deux implémentations concrètes, Dog et Cat .

Programmation à une mise en œuvre serait :

**Chien d = nouveau
chien(); d.écorce();**

Déclarer la variable « d » comme type Dog (une implémentation concrète d'Animal) nous oblige à coder vers une implémentation concrète.

Mais programmation à une interface/supertypes serait :

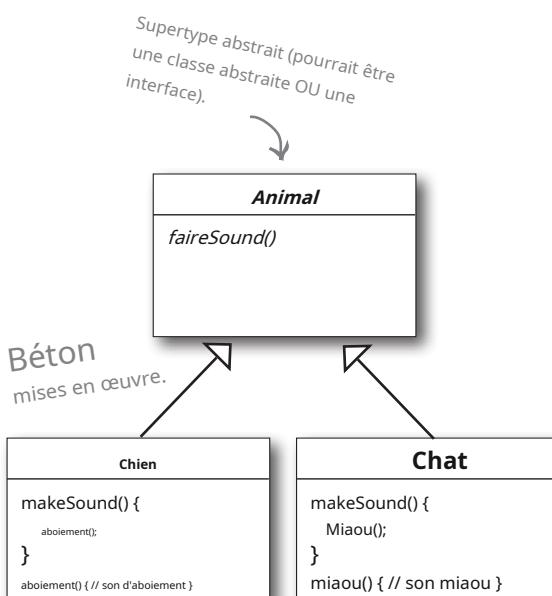
**Animal animal = nouveau
chien(); animal.makeSound();**

On sait que c'est un chien, mais on peut maintenant utiliser la référence animale de manière polymorphe.

Mieux encore, plutôt que de coder en dur linstanciation du sous-type (comme new Dog()) dans le code, **assigner l'objet d'implémentation concret au moment de l'exécution :**

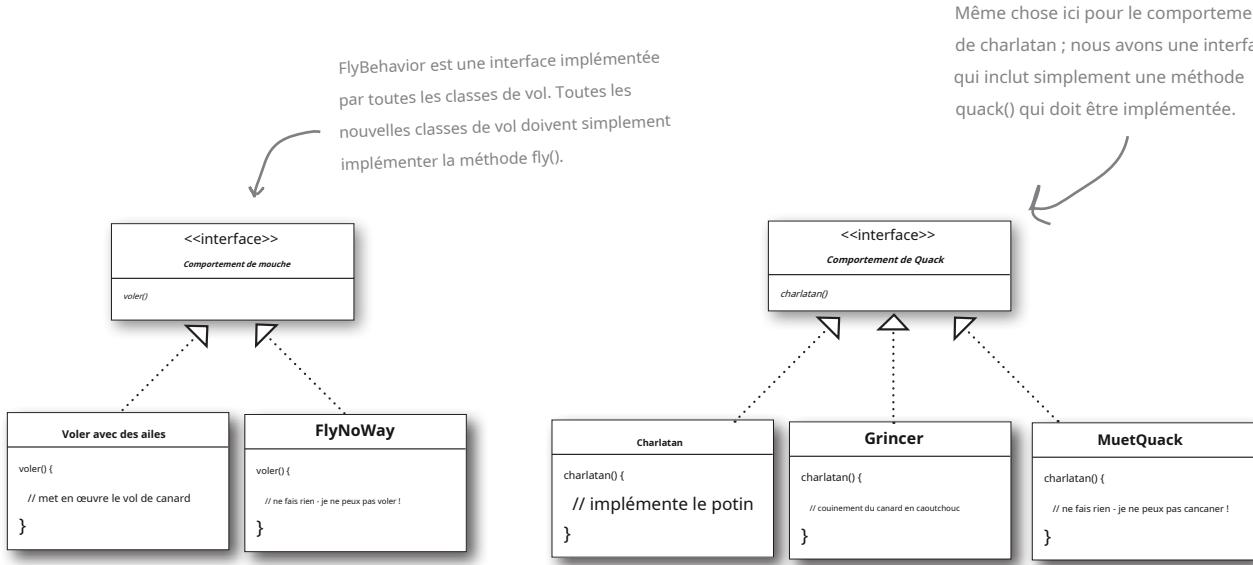
**a = obtenirAnimal();
a.makeSound();**

Nous ne savons pas QUEL est le sous-type animal réel... tout ce qui nous importe, c'est qu'il sache comment répondre à makeSound().



Mise en œuvre des comportements de canard

Nous avons ici les deux interfaces, FlyBehavior et QuackBehavior, ainsi que les classes correspondantes qui implémentent chaque comportement concret :



Voici la mise en œuvre du vol pour tous les canards qui ont des ailes.

Et voici l'implémentation pour tous les canards qui ne savent pas voler.

Même chose ici pour le comportement de charlatan ; nous avons une interface qui inclut simplement une méthode quack() qui doit être implémentée.

Avec cette conception, d'autres types d'objets peuvent réutiliser nos comportements de mouche et de charlatan parce que ces comportements ne sont plus cachés dans nos classes de canard !

Et nous pouvons ajouter de nouveaux comportements sans modifier aucune de nos classes de comportement existantes ni toucher aux classes Duck qui usage comportements de vol.

Nous bénéficions ainsi des avantages de la RÉUTILISATION sans tout le bagage qui accompagne l'héritage.

there are no Dumb Questions

Question : Devez-je toujours implémenter mon application en premier, voir où les choses changent, puis revenir en arrière pour séparer et encapsuler ces choses ?

UN:

Pas toujours ; souvent, lorsque vous concevez une application, vous anticipez les domaines qui vont varier, puis vous intégrez la flexibilité nécessaire pour les gérer dans votre code. Vous constaterez que les principes et les modèles peuvent être appliqués à n'importe quelle étape du cycle de vie du développement.

Question : Faut-il faire de Duck une interface aussi ?

UN:

Pas dans ce cas. Comme vous le verrez une fois que nous aurons tout connecté, nous bénéficions du fait que Duck ne soit pas une interface et que des canards spécifiques, comme MallardDuck, héritent de propriétés et de méthodes communes. Maintenant que nous avons enlevé ce qui varie de l'héritage Duck, nous bénéficions des avantages de cette structure sans les problèmes.

Question : C'est un peu bizarre d'avoir un cours qui n'est qu'un comportement. Les classes ne sont-elles pas censées représenter des choses ? Les classes ne sont-elles pas censées avoir à la fois un état ET un comportement ?

UN:

Dans un système OO, oui, les classes représentent des éléments qui ont généralement à la fois un état (variables d'instance) et des méthodes. Et dans ce cas, c'est un comportement. Mais même un comportement peut toujours avoir un état et des méthodes ; un comportement de vol peut avoir des variables d'instance représentant les attributs du comportement de vol (battements d'ailes par minute, altitude maximale, vitesse, etc.).



Sharpen your pencil

1 En utilisant notre nouveau design, que feriez-vous si vous aviez besoin d'ajouter le vol propulsé par fusée à l'application SimUDuck ?

2 Pouvez-vous penser à une classe qui pourrait vouloir utiliser le comportement Quack qui n'est pas un canard ?

appareil qui fait des sons de canard).
2 Un exemple, un appel de canard (un

l'interface FlyBehavior.

FlyRocketPowered qui implemente

1) Créer une classe

Réponses :

Intégrer les comportements du canard

Voici la clé : un canard va maintenant déléguer ses comportements de vol et de charlatan, au lieu d'utiliser les méthodes de charlatan et de vol définies dans la classe (ou sous-classe) Duck.

Voici comment faire :

1 Nous allons d'abord ajouter deux variables d'instance de type FlyBehavior et QuackBehavior :

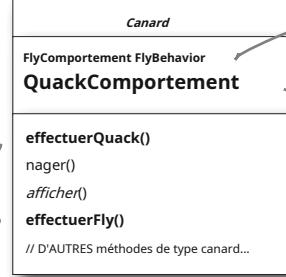
appelons-les flyBehavior et quackBehavior. Chaque objet canard en béton attribuera à ces variables un spécifique comportement au moment de l'exécution, comme FlyWithWings pour voler et Squeak pour le charlatan.

Nous supprimerons également les méthodes fly() et quack() de la classe Duck (et de toutes ses sous-classes) parce que nous avons déplacé ce comportement vers les classes FlyBehavior et QuackBehavior.

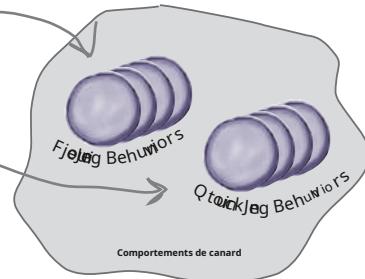
Nous remplacerons fly() et quack() dans la classe Duck par deux méthodes similaires, appelées performFly() et performQuack(); vous verrez comment ils fonctionnent ensuite.

Les variables de comportement sont déclarées comme type INTERFACE de comportement.

Ces méthodes remplacent fly() et charlatan().



Les variables d'instance contiennent une référence à un comportement spécifique au moment de l'exécution.



2 Maintenant, on implante performQuack() :

```

classe abstraite publique Canard {
    QuackBehavior
    charlatanBehavior; // plus

    public void performQuack() {
        quackBehavior.quack();
    }
}

```

Chaque Duck a une référence à quelque chose qui implémente l'interface QuackBehavior.

Plutôt que de gérer le comportement de charlatan lui-même, l'objet Duck délègue ce comportement à l'objet référencé par quackBehavior.

Assez simple, hein ? Pour faire le charlatan, un canard demande simplement à l'objet référencé par quackBehavior de charlatan. Dans cette partie du code, on s'en fout quel genre d'objet le Canard en béton est, **tout ce qui nous intéresse, c'est qu'il sache charlataner()** !

Plus d'intégration...

- 3 Ok, c'est le temps de s'en faire **comment** les variables d'instance flyBehavior et quackBehavior **sont définis**. Jetons un coup d'œil à la classe MallardDuck :

la classe publique MallardDuck étend Duck {

```
public Canard Colvert() {  
    quackBehavior = new Quack();  
    flyBehavior = new FlyWithWings();  
}
```

N'oubliez pas que MallardDuck hérite des variables d'instance quackBehavior et flyBehavior de la classe Duck.

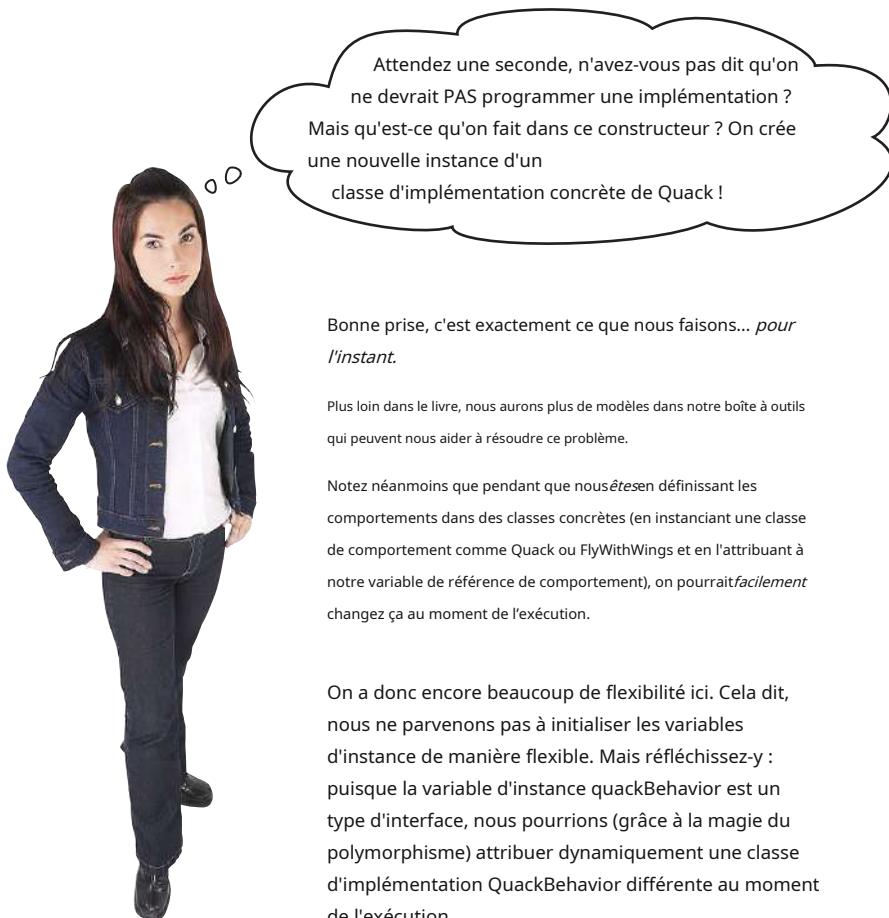
Un MallardDuck utilise la classe Quack pour gérer son charlatan, donc lorsque performQuack() est appelé, la responsabilité du charlatan est déléguée à l'objet Quack et nous obtenons un vrai charlatan.

Et il utilise FlyWithWings comme type FlyBehavior.

```
affichage public vide() {  
    System.out.println("Je suis un vrai canard colvert");  
}  
}
```

Le charlatan de MallardDuck est un vrai canard vivant **charlatan**, pas un **grincer** et pas **uncharlatan muet**. Lorsqu'un MallardDuck est instancié, son constructeur initialise la variable d'instance héritée de MallardDuck quackBehavior à une nouvelle instance de type Quack (une classe d'implémentation concrète de QuackBehavior).

Et il en va de même pour le comportement de vol du canard : le constructeur de MallardDuck initialise la variable d'instance héritée de flyBehavior avec une instance de type FlyWithWings (une classe d'implémentation concrète de FlyBehavior).



Prenez un moment et réfléchissez à la façon dont vous implémenteriez un canard afin que son comportement puisse changer au moment de l'exécution. (Vous verrez le code qui fait ça dans quelques pages.)

Tester le code Duck

- ➊ Tapez et compilez la classe Duck ci-dessous (Duck.java) et la classe MallardDuck de deux pages en arrière (MallardDuck.java).

```
classe abstraite publique Canard {  
  
    FlyBehavior flyBehavior;  
    QuackBehavior charlatanBehavior;  
    Canard public() {}  
  
    affichage vide abstrait public ();  
  
    public void performFly() {  
        flyBehavior.fly();  
    }  
  
    public void performQuack() {  
        quackBehavior.quack();  
    }  
  
    public void swim() {  
        System.out.println("Tous les canards flottent, même les leurre !");  
    }  
}
```

Déclarer deux références variables pour les types d'interface de comportement. Toutes les sous-classes de duck (dans le même paquet) en héritent.

Délégué à la classe de comportement.

- ➋ Tapez et compilez l'interface FlyBehavior (FlyBehavior.java) et les deux classes d'implémentation de comportement (FlyWithWings.java et FlyNoWay.java).

```
interface publique FlyBehavior {  
    public void fly();  
}
```

L'interface implémentée par toutes les classes de comportement de vol.

```
la classe publique FlyWithWings implémente FlyBehavior {  
    public void fly() {  
        System.out.println("J'veole !!");  
    }  
}
```

Mise en œuvre du comportement de vol pour les canards qui volent réellement...

```
classe publique FlyNoWay implémente FlyBehavior {  
    public void fly() {  
        System.out.println("Je ne peux pas voler");  
    }  
}
```

Mise en œuvre d'un comportement de vol pour les canards qui ne volent PAS (comme les canards en caoutchouc et les canards leurre).

Test du code Duck, suite...

- 3 Tapez et compilez l'interface QuackBehavior (QuackBehavior.java) et les trois classes d'implémentation de comportement (Quack.java, MuteQuack.java et Squeak.java).**

```
interface publique QuackBehavior {
    charlatan public vide ();
}
```

```
classe publique Quack met en œuvre QuackBehavior {
    public void charlatan() {
        System.out.println("Quack");
    }
}
```

```
classe publique MuteQuack implémente QuackBehavior {
    public void charlatan() {
        System.out.println("<< Silence >>");
    }
}
```

```
classe publique Squeak implémente QuackBehavior {
    public void charlatan() {
        System.out.println("Squeak");
    }
}
```

- 4 Tapez et compilez la classe de test (MiniDuckSimulator.java).**

```
classe publique MiniDuckSimulator {
    public static void main (String[] arguments) {
        Canard colvert = new MallardDuck();
        colvert.performQuack();
        colvert.performFly();
    }
}
```

Cela appelle la méthode performQuack() héritée du Canard Mallard, qui délègue ensuite au QuackBehavior de l'objet (c'est-à-dire qu'elle appelle quack() sur la référence quackBehavior héritée du canard).

Ensuite, on fait la même chose avec la méthode performFly() héritée de MallardDuck.

- 5 Exécutez le code !**



```
Aide de la fenêtre d'édition de fichier Yadayadaya
%java MiniDuckSimulator
Quack
Je vole !!
```

Définir un comportement de manière dynamique

Quel dommage d'avoir tout ce talent dynamique intégré à nos canards et de ne pas l'utiliser !

Imaginez que vous vouliez définir le type de comportement du canard via une méthode de définition sur la classe Duck, plutôt qu'en l'instanciant dans le constructeur du canard.

1) ajoutez deux nouvelles méthodes à la classe Duck :

```
public void setFlyBehavior (FlyBehavior fb) {
    flyBehavior = fb;
}

public void setQuackBehavior (QuackBehavior qb) {
    charlatanComportement = qb;
}
```

On peut appeler ces méthodes chaque fois qu'on veut changer le comportement d'un canard à la volée.

Canard
FlyComportement flyComportement
QuackComportement charlatanComportement
nager()
afficher()
effectuerQuack()
effectuerFly()
setFlyBehavior()
setQuackBehavior()
// D'AUTRES méthodes de type canard...

Note de l'éditeur : jeu de mots gratuit - correctif

2) Créez un nouveau type Duck (ModelDuck.java).

```
la classe publique ModelDuck étend Duck {
    public ModelDuck() {
        flyBehavior = new FlyNoWay();
        quackBehavior = new Quack();
    }

    affichage public vide() {
        System.out.println("Je suis un canard modèle");
    }
}
```

Notre canard modèle commence sa vie au sol... sans moyen de voler.

3) Créez un nouveau type FlyBehavior (FlyRocketPowered.java).

C'est correct, on crée un comportement de vol propulsé par une fusée.

```
la classe publique FlyRocketPowered met en œuvre FlyBehavior {
    public void fly() {
        System.out.println("J'veole avec une fusée !");
    }
}
```



- 4 Modifiez la classe de test (MiniDuckSimulator.java), ajoutez le ModelDuck et rendez le ModelDuck compatible avec les fusées.

```
classe publique MiniDuckSimulator {
    public static void main(String[] args) {
        Canard colvert = nouveau MallardDuck();
        colvert.performQuack();
        colvert.performFly();
```

```
Modèle de canard = nouveau ModelDuck();
modèle.performFly();
modèle.setFlyBehavior(nouveau FlyRocketPowered());
modèle.performFly();
```

Si cela fonctionnait, le canard modèle changerait dynamiquement son comportement de vol ! Vous ne pouvez pas faire CELA si l'implémentation se trouve à l'intérieur de la classe Duck.

- 5 Cours-le !

Aide sur la fenêtre d'édition de fichiers Yabbadabadoo

Simulateur de MiniDuck %java

Charlatan

Je vole !!

Je ne peux pas voler

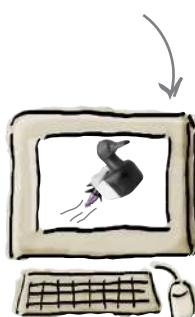
Je vole avec une fusée !



Avant

Le premier appel à performFly() délègue à l'objet flyBehavior défini dans le constructeur de ModelDuck, qui est une instance FlyNoWay.

Cela invoque la méthode de définition du comportement hérité du modèle, et... voilà ! Le modèle est soudainement doté de la capacité de voler à propulsion par fusée !



Après

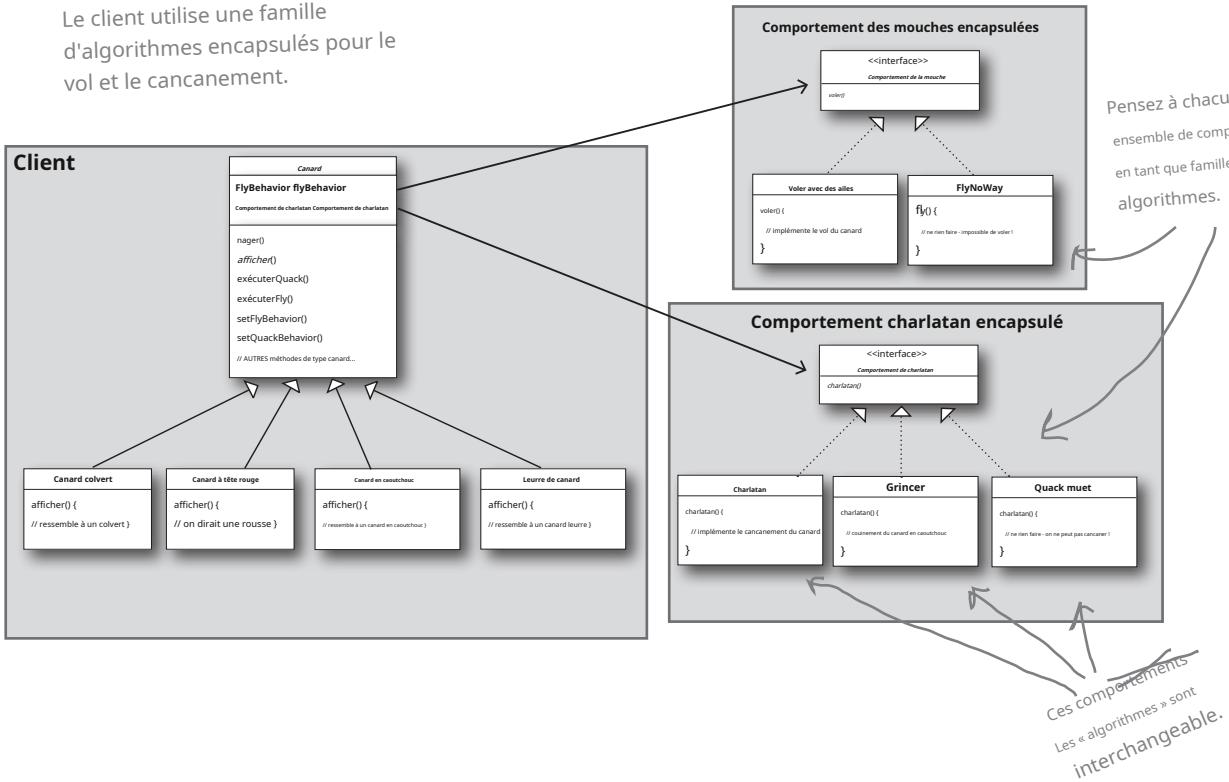
Pour modifier le comportement d'un canard au moment de l'exécution,appelez simplement la méthode setter du canard pour ce comportement.

Le point sur les comportements encapsulés

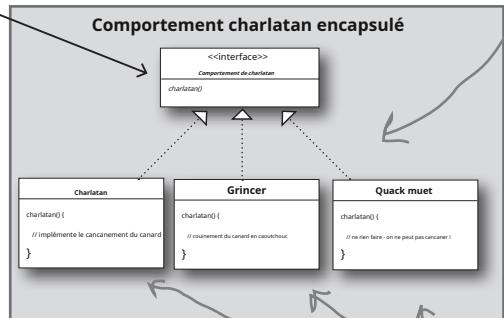
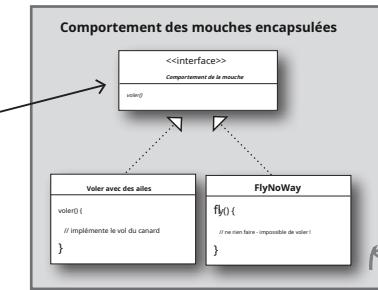
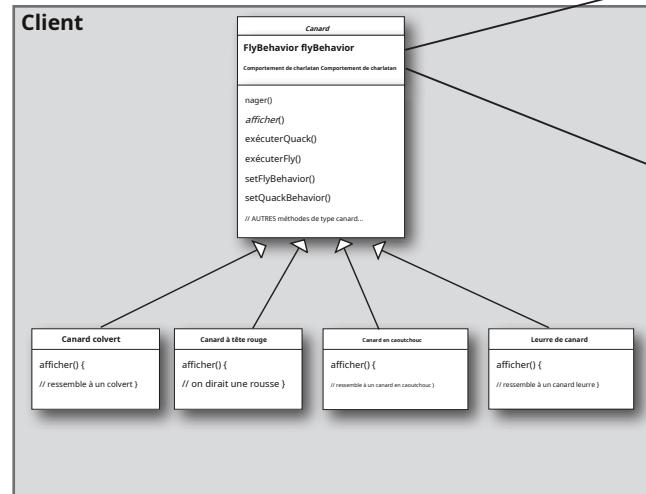
Ok, maintenant que nous avons fait une plongée en profondeur dans la conception du simulateur de canard, il est temps de remonter à la surface et d'avoir une vue d'ensemble.

Vous trouverez ci-dessous la structure complète de la classe retravaillée. Nous avons tout ce que vous attendez : des canards qui étendent Duck, des comportements de vol qui implémentent FlyBehavior et des comportements de cancan qui implémentent QuackBehavior.

Notez également que nous avons commencé à décrire les choses un peu différemment. Au lieu de considérer les comportements des canards comme un *ensemble de comportements*, nous commencerons à les considérer comme une *famille d'algorithmes*. Pensez-y : dans la conception de SimUDuck, les algorithmes représentent des choses qu'un canard ferait (différentes manières de cancaner ou de voler), mais nous pourrions tout aussi bien utiliser les mêmes techniques pour un ensemble de classes qui implémentent les moyens de calculer la taxe de vente de l'État par différents États.

Faites très attention à la *relation* entre les classes. En fait, prenez votre stylo et écrivez la relation appropriée (IS-A, HAS-A et IMPLEMENTS) sur  Assurez-vous de le faire.

Le client utilise une famille d'algorithmes encapsulés pour le vol et le cancanement.



Pensez à chacun ensemble de comportements en tant que famille de algorithmes.

Ces comportements Les « algorithmes » sont interchangeables.

HAS-A peut être meilleur que IS-A

La relation HAS-A est intéressante : chaque canard a un FlyBehavior et un QuackBehavior auxquels il délègue le vol et le cancanement.

Lorsque vous mettez deux classes ensemble comme ceci, vous utilisez **composition**. Au lieu de hériter leur comportement, les canards obtiennent leur comportement en étant composé avec le bon objet de comportement.

Il s'agit d'une technique importante ; en fait, elle constitue la base de notre troisième principe de conception :



Principe de conception

Privilégiez la composition plutôt que l'héritage.

Comme vous l'avez vu, la création de systèmes à l'aide de la composition vous offre beaucoup plus de flexibilité. Non seulement cela vous permet d'encapsuler une famille d'algorithmes dans leur propre ensemble de classes, mais cela vous permet également **modifier le comportement lors de l'exécution** tant que l'objet avec lequel vous composez implémente l'interface de comportement correcte.

La composition est utilisée dans *beaucoup* modèles de conception et vous en verrez beaucoup plus sur ses avantages et ses inconvénients tout au long du livre.



Un appeau de canard est un dispositif que les chasseurs utilisent pour imiter les cris (coin-coins) des canards.
Comment mettriez-vous en œuvre votre propre appeau de canard qui ne pashériter de la classe Canard ?



Gourou et étudiant...

Gourou:Dites-moi ce que vous avez appris des méthodes orientées objet.

Étudiant:Gourou, j'ai

J'ai appris que la promesse de la méthode orientée objet est la réutilisation.

Gourou:Continuer...

Étudiant:Gourou, grâce à l'héritage, toutes les bonnes choses peuvent être réutilisées et nous parvenons ainsi à réduire considérablement le temps de développement, comme nous coupons rapidement le bambou dans les bois.

Gourou:Est-ce que plus de temps est consacré au code **avant ou après** le développement est terminé ?

Étudiant:La réponse est **après**, Guru. Nous passons toujours plus de temps à maintenir et à modifier les logiciels qu'à les développer.

Gourou:Alors, faut-il faire des efforts pour réutiliser **au-dessus** de maintenabilité et extensibilité ?

Étudiant:Gourou, je crois qu'il y a du vrai là-dedans.

Gourou:Je vois que tu as encore beaucoup à apprendre. J'aimerais que tu approfondisses la question de l'héritage. Comme tu l'as vu, l'héritage a ses problèmes et il existe d'autres moyens de parvenir à la réutilisation.

En parlant de modèles de conception...



Félicitations pour votre premier modèle !

Vous venez de postulervotre premier modèle de conception—leSTRATÉGIE Modèle. C'est vrai, vous avez utilisé le modèle de stratégie pour retravailler l'application SimUDuck.
Grâce à ce modèle, le simulateur est prêt pour tous les changements que ces dirigeants pourraient envisager lors de leur prochain voyage d'affaires à Maui.

Maintenant que nous vous avons fait prendre le long chemin pour l'apprendre, voici la définition formelle de ce modèle :

Le modèle de stratégiedéfinit une famille d'algorithmes, encapsule chacun d'eux et les rend interchangeables. La stratégie permet à l'algorithme de varier indépendamment des clients qui l'utilisent.

Utilisez CETTE définition lorsque vous avez besoin d'impressionner vos amis et d'influencer les principaux dirigeants.



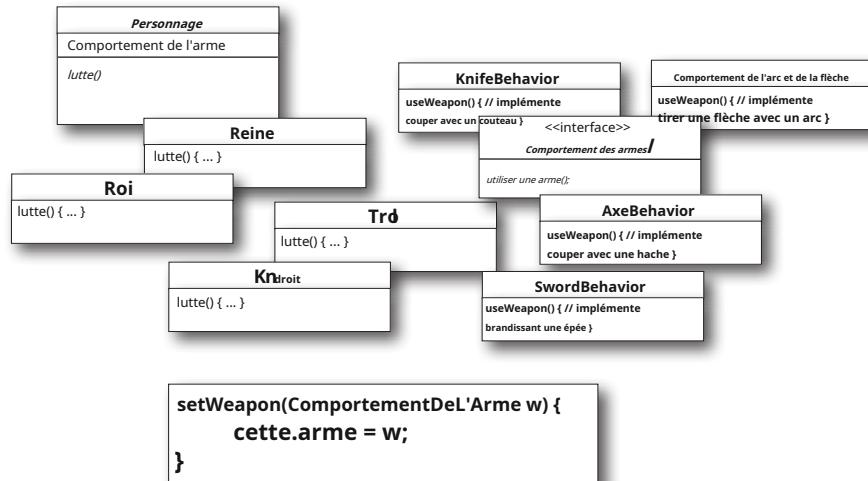
Puzzle de conception

Vous trouverez ci-dessous un tas de classes et d'interfaces pour un jeu d'action-aventure. Vous trouverez des classes pour les personnages du jeu ainsi que des classes pour les comportements d'armes que les personnages peuvent utiliser dans le jeu. Chaque personnage peut utiliser une arme à la fois, mais peut changer d'arme à tout moment pendant le jeu. Votre travail consiste à tout régler...

(Les réponses sont à la fin du chapitre.)

Votre tâche:

1. Organiser les classes.
2. Identifiez une classe abstraite, une interface et huit classes.
3. Dessinez des flèches entre les classes.
 - a. Dessinez ce type de flèche pour l'héritage (« s'étend »).
 - b. Dessinez ce type de flèche pour l'interface (« implements »).
 - c. Dessinez ce type de flèche pour HAS-A.
4. Placez la méthode setWeapon() dans la bonne classe.



Entendu au restaurant local...



Quelle est la différence entre ces deux commandes ? Aucune ! Elles correspondent toutes les deux à la même commande, sauf qu'Alice utilise deux fois plus de mots et met à l'épreuve la patience d'une cuisinière grincheuse qui prépare des plats rapides.

Qu'est-ce que Flo a qu'Alice n'a pas ?**Un vocabulaire partagé** avec le cuisinier de la restauration rapide. Non seulement cela facilite la communication avec le cuisinier, mais cela lui donne aussi moins de choses à retenir car il a tous les modèles de restauration dans sa tête.

Les modèles de conception vous offrent un vocabulaire partagé avec d'autres développeurs. Une fois que vous avez acquis le vocabulaire, vous pouvez plus facilement communiquer avec d'autres développeurs et inspirer ceux qui ne connaissent pas les modèles à commencer à les apprendre. Cela améliore également votre réflexion sur les architectures en vous permettant **pense à la modèle niveau**, pas les détails *objets* niveau.

Entendu dans la cabine d'à côté...



Pouvez-vous penser à d'autres vocabulaires partagés qui sont utilisés au-delà de la conception OO et du langage courant au restaurant ? (Indice : qu'en est-il des mécaniciens automobiles, des charpentiers, des chefs gastronomiques et des contrôleurs aériens ?) Quelles qualités sont communiquées avec le jargon ?

Pouvez-vous penser à des aspects de la conception OO qui sont communiqués avec les noms de modèles ? Quelles qualités sont communiquées avec le nom « Modèle de stratégie » ?

La puissance d'un vocabulaire de modèles partagés

Lorsque vous communiquez en utilisant des modèles, vous faites bien plus que simplement partager du jargon.

Les vocabulaires de modèles partagés sont PUISSANTS. Lorsque vous communiquez avec un autre développeur ou votre équipe à l'aide de modèles, vous communiquez non seulement un nom de modèle, mais un ensemble complet de qualités, de caractéristiques et de contraintes que le modèle représente.

« Nous utilisons le modèle de stratégie pour implémenter les différents comportements de nos canards. » Cela vous indique que le comportement du canard a été encapsulé dans son propre ensemble de classes qui peuvent être facilement étendues et modifiées, même au moment de l'exécution si nécessaire.

Les modèles vous permettent d'en dire plus avec moins. Lorsque vous utilisez un modèle dans une description, les autres développeurs connaissent rapidement et précisément le design que vous avez en tête.

Parler au niveau du modèle vous permet de rester « dans la conception » plus longtemps. Parler de systèmes logiciels à l'aide de modèles vous permet de maintenir la discussion au niveau de la conception, sans avoir à plonger dans les détails de l'implémentation d'objets et de classes.

À combien de réunions de conception avez-vous participé qui se sont rapidement transformées en détails de mise en œuvre ?

Les vocabulaires partagés peuvent dynamiser votre équipe de développement. Une équipe bien familiarisée avec les modèles de conception peut évoluer plus rapidement avec moins de place pour les malentendus.

Au fur et à mesure que votre équipe commence à partager des idées de conception et des expériences en termes de modèles, vous créerez une communauté d'utilisateurs de modèles.

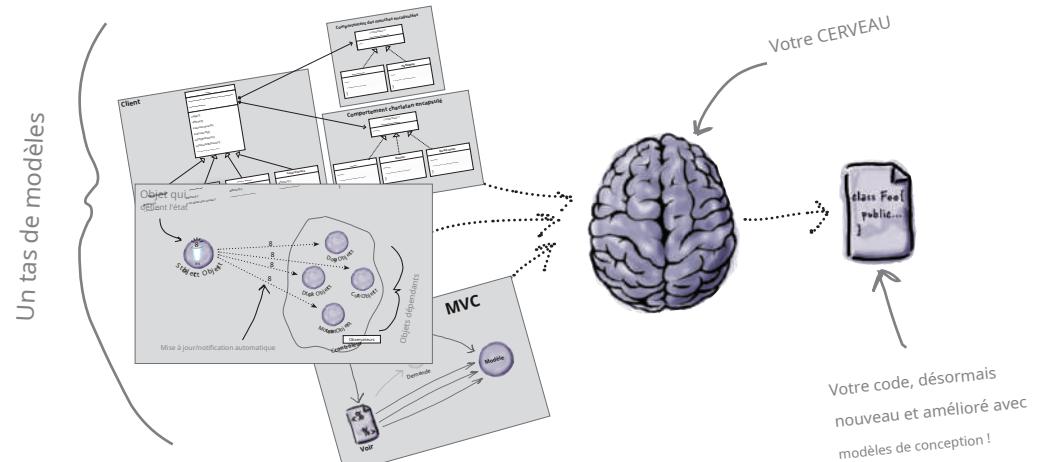
Les vocabulaires partagés encouragent les développeurs plus juniors à se mettre à niveau. Les développeurs juniors admirent les développeurs expérimentés. Lorsque les développeurs seniors utilisent des modèles de conception, les développeurs juniors sont également motivés à les apprendre. Créez une communauté d'utilisateurs de modèles au sein de votre organisation.

Pensez à créer un groupe d'étude des modèles au sein de votre organisation. Vous pourriez même être rémunéré pendant que vous apprenez...

Comment utiliser les modèles de conception ?

Nous avons tous déjà utilisé des bibliothèques et des frameworks prêts à l'emploi. Nous les prenons, écrivons du code en fonction de leurs API, les compilons dans nos programmes et bénéficiions d'une grande quantité de code écrit par quelqu'un d'autre. Pensez aux API Java et à toutes les fonctionnalités qu'elles vous offrent : réseau, interface graphique, E/S, etc. Les bibliothèques et les frameworks contribuent grandement à un modèle de développement dans lequel nous pouvons simplement choisir des composants et les intégrer directement. Mais... ils ne nous aident pas à structurer nos propres applications de manière plus facile à comprendre, plus facile à maintenir et plus flexible. C'est là que les modèles de conception entrent en jeu.

Les modèles de conception ne sont pas intégrés directement dans votre code, ils sont d'abord intégrés dans votre CERVEAU. Une fois que vous avez intégré à votre cerveau une bonne connaissance pratique des modèles, vous pouvez commencer à les appliquer à vos nouvelles conceptions et à retravailler votre ancien code lorsque vous constatez qu'il se dégrade en un fouillis inflexible.



*there are no
Dumb Questions*

Q:

Si les modèles de conception sont si géniaux, pourquoi quelqu'un ne pourrait-il pas créer une bibliothèque à partir d'eux pour que je n'aie pas à les faire ?

UN:

Les modèles de conception sont de niveau supérieur aux bibliothèques. Ils nous indiquent comment structurer les classes et les objets pour résoudre certains problèmes, et il nous incombe d'adapter ces modèles pour qu'ils correspondent à notre application particulière.

Q:

Les bibliothèques et les frameworks ne sont-ils pas également des modèles de conception ?

UN:

Les frameworks et les bibliothèques ne sont pas des modèles de conception. Ils fournissent des implémentations spécifiques que nous relierons à notre code. Parfois, cependant, les bibliothèques et les frameworks utilisent des modèles de conception dans leurs implémentations. C'est formidable, car une fois que vous aurez compris les modèles de conception, vous comprendrez plus rapidement les API structurées autour de modèles de conception.

Q:

Donc, il n'existe pas de bibliothèques de modèles de conception ?

UN:

Non, mais vous en apprendrez plus tard sur les catalogues de modèles avec des listes de modèles que vous pouvez appliquer à vos applications.



Développeur sceptique



Guru des modèles conviviaux

Promoteur:D'accord, hmm, mais n'est-ce pas tout simplement une bonne conception orientée objet ? Je veux dire, tant que je suis l'encapsulation et que je connais l'abstraction, l'héritage et le polymorphisme, dois-je vraiment réfléchir aux modèles de conception ? N'est-ce pas assez simple ? N'est-ce pas la raison pour laquelle j'ai suivi tous ces cours OO ? Je pense que les modèles de conception sont utiles pour les personnes qui ne connaissent pas la bonne conception OO.

Gourou:Ah, c'est l'un des vrais malentendus du développement orienté objet : en connaissant les bases de l'OO, nous serons automatiquement capables de construire des systèmes flexibles, réutilisables et maintenables.

Promoteur:Non?

Gourou:Non. Il s'avère que la construction de systèmes OO dotés de ces propriétés n'est pas toujours évidente et n'a été découverte que grâce à un travail acharné.

Promoteur:Je crois que je commence à comprendre. Ces manières, parfois peu évidentes, de construire des systèmes orientés objet ont été rassemblées...

Gourou:...oui, dans un ensemble de modèles appelés modèles de conception.

Promoteur:Ainsi, en connaissant les modèles, je peux éviter le travail difficile et passer directement aux modèles qui fonctionnent toujours ?

Gourou:Oui, dans une certaine mesure, mais n'oubliez pas que le design est un art. Il y aura toujours des compromis. Mais si vous suivez des modèles de conception bien pensés et éprouvés, vous aurez une longueur d'avance.

Promoteur:Que dois-je faire si je ne trouve pas de modèle ?



N'oubliez pas que la connaissance de concepts tels que l'abstraction, l'héritage et le polymorphisme ne fait pas de vous un bon concepteur orienté objet. Un gourou du design réfléchit à la manière de créer des conceptions flexibles, maintenables et capables de s'adapter aux changements.

Gourou:Certains principes orientés objet sous-tendent les modèles, et les connaître vous aidera à faire face lorsque vous ne parvenez pas à trouver un modèle correspondant à votre problème.

Promoteur:Des principes ? Vous voulez dire au-delà de l'abstraction, de l'encapsulation et...

Gourou:Oui, l'un des secrets de la création de systèmes OO maintenables est de réfléchir à la manière dont ils pourraient évoluer à l'avenir, et ces principes abordent ces problèmes.



Des outils pour votre boîte à outils de conception

Vous avez presque terminé le premier chapitre ! Vous avez déjà ajouté quelques outils à votre boîte à outils OO ; dressons-en la liste avant de passer au chapitre 2.

Notions de base sur OO

- Abstraction
- Encapsulation
- Polymorphisme
- Héritage

Nous partons du principe que vous connaissez les bases de OO comme l'abstraction, l'encapsulation, le polymorphisme et l'héritage. Si vous êtes un peu rouillé sur ces points, sortez votre livre orienté objet préféré et révisez-le, puis parcourez à nouveau ce chapitre.

Principes OO

Encapsuler ce qui varie.

Privilégier la composition plutôt que héritage.

Programmez vers des interfaces, pas vers des implémentations.

Nous les examinerons de plus près plus tard et en ajouterons quelques autres à la liste.

Modèles OO

Stratégie - définit une famille d'algorithmes, encapsule chacun d'eux et les rend interchangeables. La stratégie permet à l'algorithme de varier indépendamment des clients qui l'utilisent.

Un de terminé, il en reste encore beaucoup à faire !



BULLET POINTS

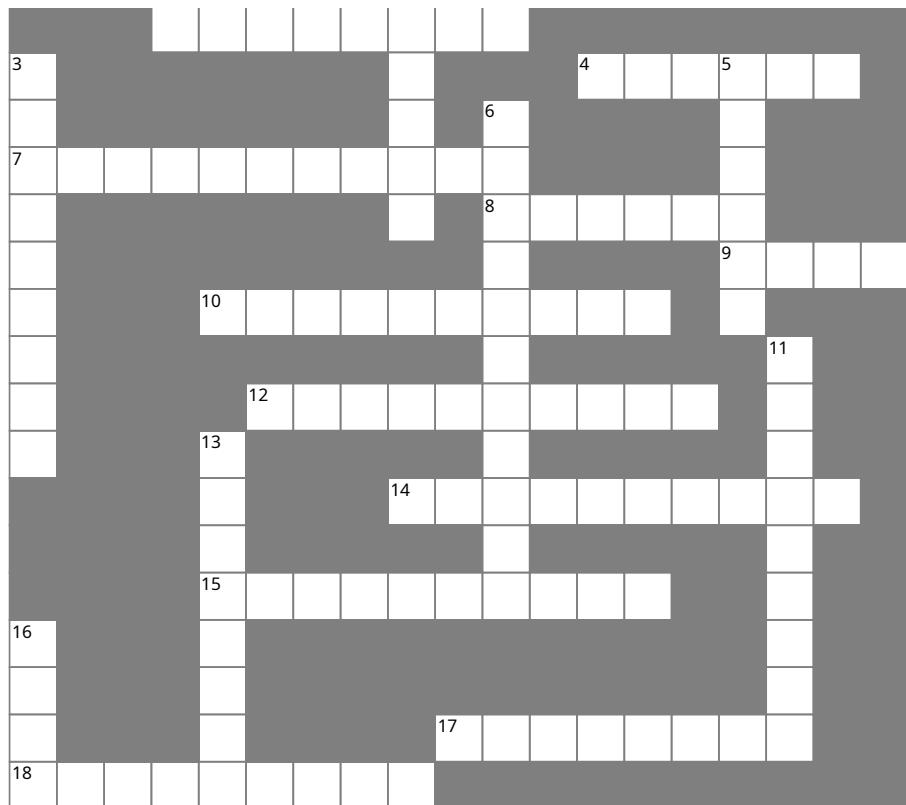
- f Connaitre les bases OO ne fait pas de vous un bon concepteur OO.
- f Les bonnes conceptions OO sont réutilisables, extensibles et maintenables.
- f Les modèles vous montrent comment créer des systèmes dotés de bonnes qualités de conception OO.
- f Les modèles sont prouvés orienté objet expérience.
- f Les modèles ne vous donnent pas de code, ils vous donnent des solutions générales à problèmes de conception. Vous appliquez-les à votre application spécifique.
- f Les modèles ne sont pas inventé, ils sont découvert.
- f La plupart des modèles et Les principes abordent les questions de changement dans le logiciel.
- f La plupart des modèles permettent à une partie d'un système de varier indépendamment de toutes les autres parties.
- f Nous essayons souvent de prendre ce qui varie dans un système et de l'encapsuler.
- f Les modèles fournissent un langage partagé qui peut maximiser la valeur de votre communication avec d'autres développeurs.



Mots croisés sur les modèles de conception

Donnons quelque chose à faire à votre cerveau droit.

C'est votre jeu de mots croisés standard ; tous les mots de solution proviennent de ce chapitre.



À TRAVERS

1. Les modèles peuvent nous aider à créer des applications _____.
4. Les stratégies peuvent être _____.
7. Privilégiez cela plutôt que l'héritage.
8. Constante de développement.
9. Java IO, réseau, son.
10. La plupart des modèles découlent de OO _____.
12. Les modèles de conception sont un _____ partagé.
14. Bibliothèques de haut niveau.
15. Apprenez des _____ de l'autre.
17. Modèle qui a corrigé le simulateur.
18. Programmez ceci, pas une implémentation.

- ### VERS LE BAS
2. Les motifs entrent dans votre _____.
 3. Canard qui ne sait pas cancaner.
 5. Les canards en caoutchouc font un _____.
 6. _____ ce qui varie.
 11. Croque-fromage au bacon.
 13. Rick était ravi de ce modèle.
 16. La démonstration de canard se trouvait ici.

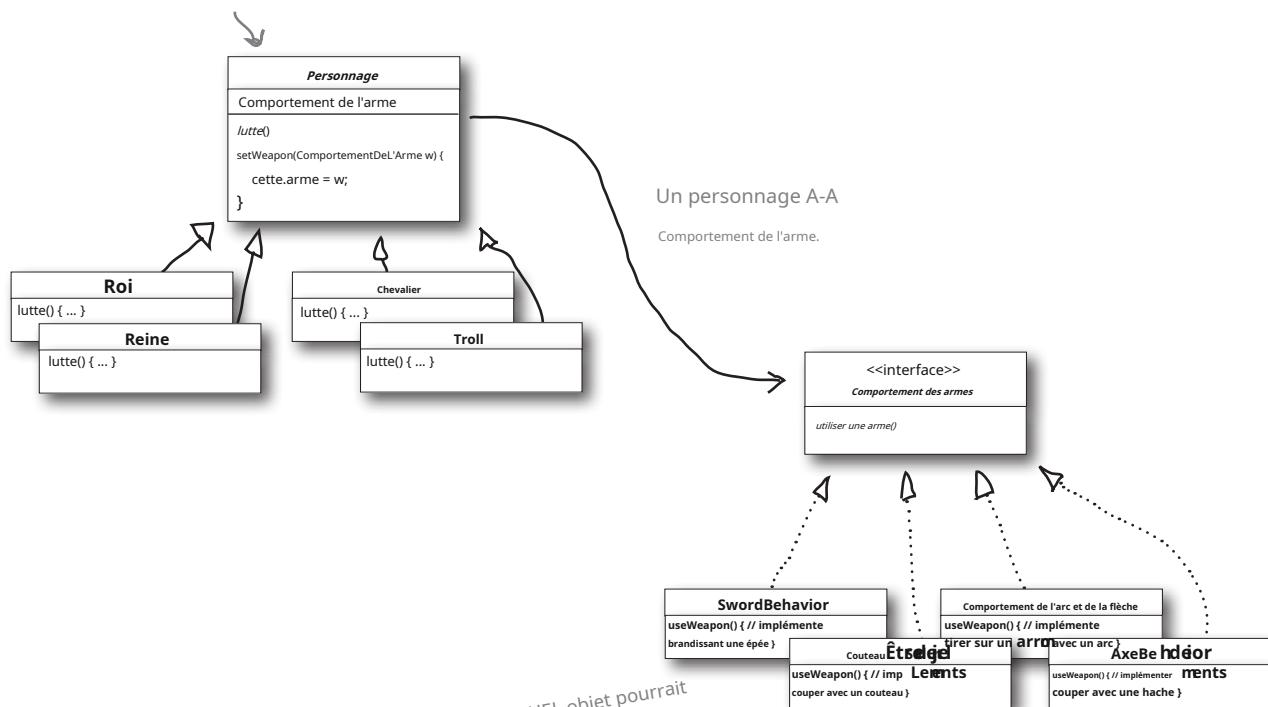


Solution du puzzle de conception

Character est la classe abstraite pour tous les autres personnages (Roi, Reine, Chevalier et Troll), tandis que WeaponBehavior est une interface que tous les comportements d'armes implémentent. Ainsi, tous les personnages et armes réels sont des classes concrètes.

Pour changer d'arme, chaque personnage appelle la méthode setWeapon(), qui est définie dans la superclasse Character. Pendant un combat, la méthode useWeapon() est appelée sur l'arme actuelle d'un personnage donné pour infliger de gros dégâts corporels à un autre personnage.

Abstrait



Notez que N'IMPORTE QUEL objet pourrait implémenter l'interface WeaponBehavior, par exemple un trombone, un tube de dentifrice ou un bar muté.



Sharpen your pencil Solution

Parmi les éléments suivants, lesquels constituent des inconvénients liés à l'utilisation de sous-classes pour fournir un comportement Duck spécifique ? (Choisissez toutes les réponses applicables.) Voici notre solution.

- A. Le code est dupliqué dans les sous-classes.
- B. Les changements de comportement lors de l'exécution sont difficiles.
- C. On ne peut pas faire danser les canards.
- D. Il est difficile d'acquérir des connaissances sur tous les comportements des canards.
- E. Les canards ne peuvent pas voler et cancaner en même temps.
- F. Les changements peuvent affecter involontairement d'autres canards.



Sharpen your pencil Solution

Quels sont les facteurs qui influent sur le changement de vos applications ? Votre liste est peut-être très différente, mais voici quelques-uns des nôtres. Cela vous semble familier ? Voici notre solution.

Mes clients ou utilisateurs décident qu'ils veulent autre chose ou qu'ils veulent de nouvelles fonctionnalités.

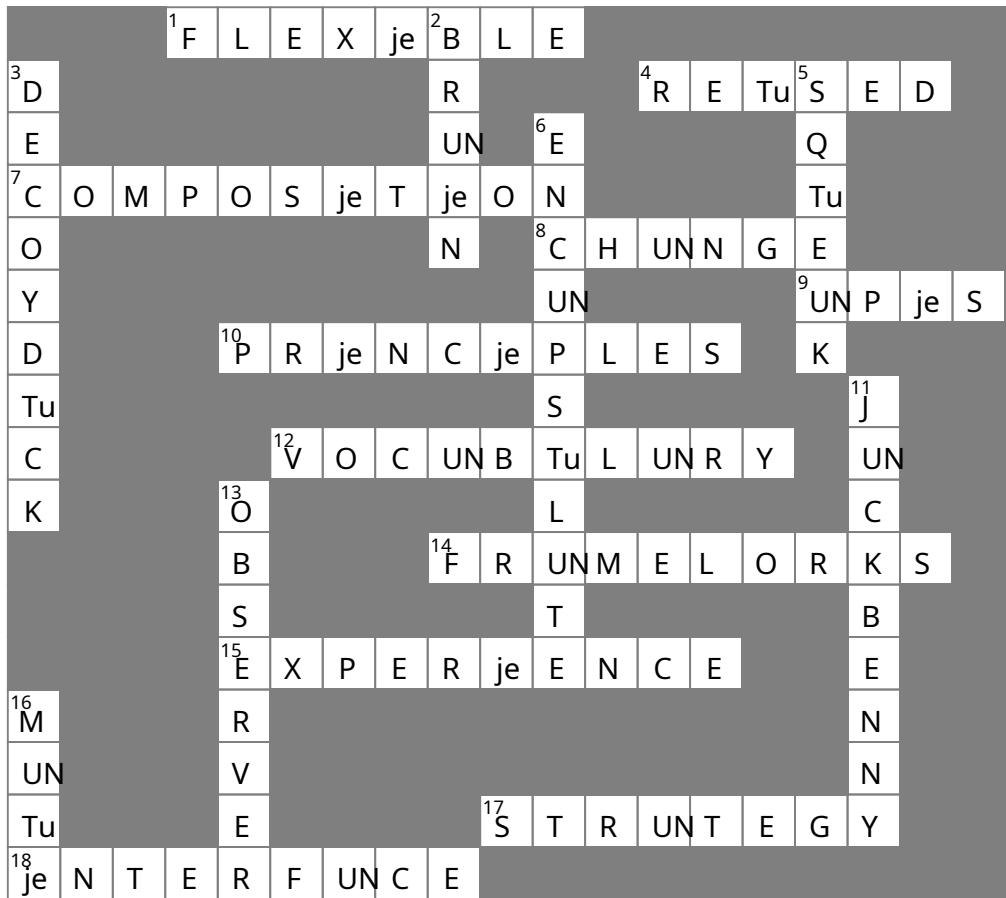
Mon entreprise a décidé de faire appel à un autre fournisseur de bases de données et achète également ses données auprès d'un autre fournisseur qui utilise un format de données différent. Argh !

Eh bien, la technologie évolue et nous devons mettre à jour notre code pour utiliser les protocoles.

Nous avons suffisamment appris en construisant notre système pour que nous souhaitions revenir en arrière et faire les choses un peu mieux.



Solution de mots croisés sur les modèles de conception



de modèle d'observateur

Garder votre Objets à connaître



Hé Jerry, je préviens tout le monde
que la réunion du groupe Patterns a
été déplacée à samedi soir.
Nous allons parler du modèle de
Observateur. Ce modèle est
le Le meilleur ! C'est le MEILLEUR, Jerry !

You ne voulez pas manquer quelque chose d'intéressant

Ça arrive, et toi ? Nous avons un modèle qui maintient vos objets au courant quand quelque chose qu'ils se soucient de il se passe quelque chose. C'est le modèle Observer. C'est l'un des modèles de conception les plus couramment utilisés et il est incroyablement utile. Nous allons examiner toutes sortes d'aspects intéressants d'Observer, comme son *relations un-à-plusieurs* et *couplage lâche*. Et, avec ces concepts en tête, comment pouvez-vous ne pas être l'âme de la Patterns Party ?

Félicitations!

Votre équipe vient de remporter le contrat de construction
de la prochaine génération de Weather-O-Rama, Inc.
base internet d Station de surveillance météorologique.



Météo-O-Rama, Inc.
100, rue Main
Allée des tornades, OK 45021

Énoncé des travaux

Félicitations pour avoir été sélectionné pour construire notre station de surveillance météorologique de nouvelle génération basée sur Internet !

La station météorologique sera basée sur notre objet WeatherData en instance de brevet, qui suit les conditions météorologiques actuelles (température, humidité et pression barométrique). Nous aimerais que vous créiez une application qui fournit initialement trois éléments d'affichage : les conditions actuelles, les statistiques météorologiques et une prévision simple, le tout mis à jour en temps réel à mesure que l'objet WeatherData acquiert les mesures les plus récentes.

De plus, il s'agit d'une station météorologique extensible. Weather-O-Rama souhaite permettre à d'autres développeurs d'écrire leurs propres affichages météo et de les intégrer directement. Il est donc important que de nouveaux affichages soient faciles à ajouter à l'avenir.

Weather-O-Rama pense que nous avons un excellent modèle économique : une fois que les clients sont convaincus, nous avons l'intention de leur facturer chaque écran qu'ils utilisent. Et maintenant, la meilleure partie : nous allons vous payer en stock-options.

Nous avons hâte de découvrir votre conception et votre application alpha.

Cordialement,

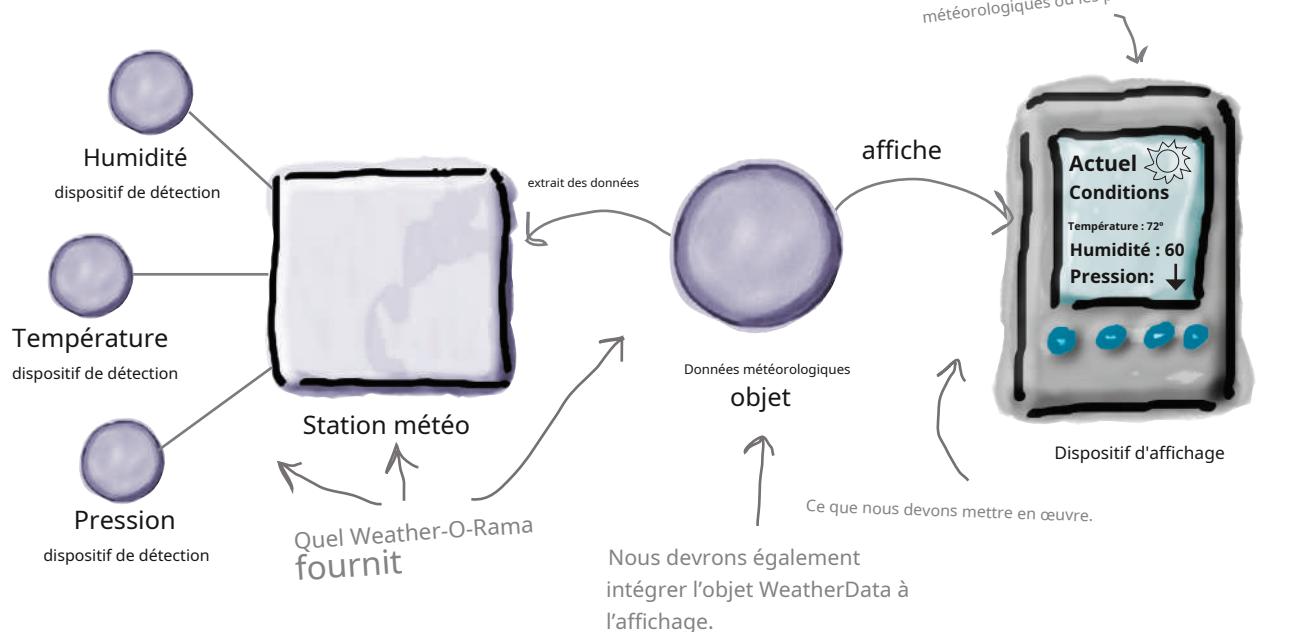
Johnny Hurricane

Johnny Hurricane, PDG

PS Voir les fichiers sources WeatherData ci-joints !

Présentation de l'application Weather Monitoring

Examinons l'application de surveillance météorologique que nous devons fournir, à la fois ce que Weather-O-Rama nous offre et ce que nous allons devoir créer ou étendre. Le système comporte trois composants : la station météorologique (le dispositif physique qui acquiert les données météorologiques réelles), l'objet WeatherData (qui suit les données provenant de la station météorologique et met à jour les affichages) et l'affichage qui montre aux utilisateurs les conditions météorologiques actuelles :



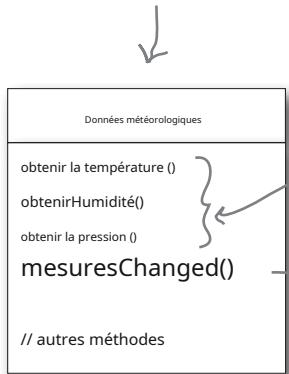
L'objet WeatherData a été écrit par Weather-O-Rama et sait comment communiquer avec la station météo physique pour obtenir des données météo mises à jour. Nous devrons adapter l'objet WeatherData pour qu'il sache comment mettre à jour l'affichage. Nous espérons que Weather-O-Rama nous a donné des conseils sur la manière de procéder dans le code source. N'oubliez pas que nous sommes responsables de la mise en oeuvre de trois éléments d'affichage différents : les conditions actuelles (affiche la température, l'humidité et la pression), les statistiques météorologiques et une simple prévision.

Notre travail, si nous choisissons de l'accepter, est donc de créer une application qui utilise l'objet WeatherData pour mettre à jour trois affichages pour les conditions actuelles, les statistiques météorologiques et les prévisions.

Déballage de la classe WeatherData

Examinons les pièces jointes du code source que Johnny Hurricane, le PDG, nous a envoyées. Nous commencerons par la classe WeatherData :

Voici notre classe WeatherData.



Ces trois méthodes renvoient les mesures météorologiques les plus récentes pour la température, l'humidité et la pression barométrique, respectivement.

Nous ne nous soucions pas actuellement de la manière dont il obtient ces données, nous savons simplement que l'objet WeatherData reçoit des informations mises à jour de la station météo.

Notez que chaque fois que WeatherData a mis à jour des valeurs, la méthode measurementsChanged() est appelée.

```
/*
 * Cette méthode est appelée
 * chaque fois que les mesures météorologiques
 * ont été mis à jour
 *
 */
public void mesureChanged() {
    // Votre code va ici
}
```

Données météorologiques.java

Notre futur-
affichage implémenté.



Dispositif d'affichage

Il semble que Weather-O-Rama ait laissé une note dans les commentaires pour ajouter notre code ici. C'est peut-être ici que nous devons mettre à jour l'affichage (une fois que nous l'aurons implémenté)

Notre travail consiste donc à modifier la méthode measurementsChanged() afin qu'elle mette à jour les trois affichages pour les conditions actuelles, les statistiques météorologiques et les prévisions.

Notre objectif

Nous savons que nous devons implémenter un affichage, puis faire en sorte que WeatherData mette à jour cet affichage chaque fois qu'il contient de nouvelles valeurs, ou, en d'autres termes, chaque fois que la méthode measurementsChanged() est appelée. Mais comment ? Réfléchissons à ce que nous essayons d'accomplir :

- Nous savons que la classe WeatherData possède des méthodes getter pour trois valeurs de mesure : température, humidité et pression barométrique.
- Nous savons que la méthode measurementsChanged() est appelée à chaque fois que de nouvelles données de mesure météorologique sont disponibles. (Encore une fois, nous ne savons pas comment cette méthode est appelée, nous savons simplement qu'elle est appelée.)
- Nous devrons implémenter trois éléments d'affichage qui utilisent les données météorologiques : *a conditions actuelles afficher*, *un affichage des statistiques*, et *un prévision affichage*. Ces affichages doivent être mis à jour aussi souvent que WeatherData comporte de nouvelles mesures.
- Pour mettre à jour les affichages, nous ajouterons du code à la méthode measurementsChanged().

Objectif extensible

Mais pensons aussi à l'avenir. Vous vous souvenez de la constante dans le développement de logiciels ? Le changement. Si la station météorologique rencontre un succès, nous prévoyons qu'il y aura plus de trois écrans à l'avenir. Alors, pourquoi ne pas créer un marché pour des écrans supplémentaires ? Alors, pourquoi ne pas intégrer :

- Extensibilité : d'autres développeurs peuvent vouloir créer de nouveaux affichages personnalisés. Pourquoi ne pas permettre aux utilisateurs d'ajouter (ou de supprimer) autant d'éléments d'affichage qu'ils le souhaitent pour l'application ? Actuellement, nous connaissons le nombre initial *trois* types d'affichage (conditions actuelles, statistiques et prévisions), mais nous nous attendons à un marché dynamique pour les nouveaux affichages à l'avenir.



Une première mise en œuvre malavisée de la station météo

Voici une première possibilité d'implémentation : comme nous l'avons vu, nous allons ajouter notre code à la méthode measurementsChanged() dans la classe WeatherData :

classe publique WeatherData {

// déclarations de variables d'instance

public void measuresChanged() {

```
float temp = getTemperature(); float  
humidité = getHumidity(); float  
pression = getPressure();
```

```
currentConditionsDisplay.update(temp, humidité, pression);  
statisticsDisplay.update(temp, humidité, pression);  
forecastDisplay.update(temp, humidité, pression);
```

}

// autres méthodes WeatherData ici

}

Voici la méthode measurementsChanged().

Et voici nos ajouts de code...

Tout d'abord, nous récupérons les mesures les plus récentes en appelant les méthodes getter de WeatherData. Nous attribuons chaque valeur à une variable nommée de manière appropriée.

Ensuite, nous allons mettre à jour chaque écran...

... en appelant sa méthode de mise à jour et en lui transmettant les mesures les plus récentes.



Sharpen your pencil

Sur la base de notre première mise en œuvre, lesquels des énoncés suivants s'appliquent ?

(Choisissez toutes les réponses applicables.)

- A. Nous codons pour du concret implémentations, pas interfaces.
- B. Pour chaque nouvel affichage, nous devrons modifier ce code.
- C. Nous n'avons aucun moyen d'ajouter (ou de supprimer) des éléments d'affichage au moment de l'exécution.
- D. Les éléments d'affichage n'implémentent pas d'interface commune.
- E. Nous n'avons pas encapsulé la partie qui change.
- F. Nous violons l'encapsulation de la classe WeatherData.

Quel est le problème avec notre implémentation ?

Repensez à tous ces concepts et principes du chapitre 1 : lesquels enfreignons-nous et lesquels ne violons-nous pas ? Pensez en particulier aux effets du changement sur ce code. Examinons notre réflexion en examinant le code :

```
public void measuresChanged()
```

Jetons un autre coup d'œil...

```
    float temp = getTemperature(); float  
    humidité = getHumidity(); float  
    pression = getPressure();
```

Cela ressemble à un domaine de changement. Nous devons le circonscrire.

```
    currentConditionsDisplay.update(temp, humidité, pression);  
    statisticsDisplay.update(temp, humidité, pression);  
    forecastDisplay.update(temp, humidité, pression);
```

}

En codant vers des implémentations concrètes, nous n'avons aucun moyen d'ajouter ou de supprimer d'autres éléments d'affichage sans apporter de modifications au code.

Au moins, nous semblons utiliser une interface commune pour communiquer avec les éléments d'affichage... ils ont tous une méthode update() qui prend les valeurs de température, d'humidité et de pression.



Et si nous voulons ajouter ou supprimer des écrans au moment de l'exécution ? Cela semble codé en dur.

Euh, je sais que je suis nouveau ici, mais étant donné que nous sommes dans le chapitre Observer Pattern, peut-être devrions-nous commencer à l'utiliser ?



Bonne idée. Jetons un œil à Observer, puis revenons et découvrons comment l'appliquer à l'application Weather Monitoring.

Découvrez le modèle Observer

Vous savez comment fonctionnent les abonnements aux journaux ou aux magazines :

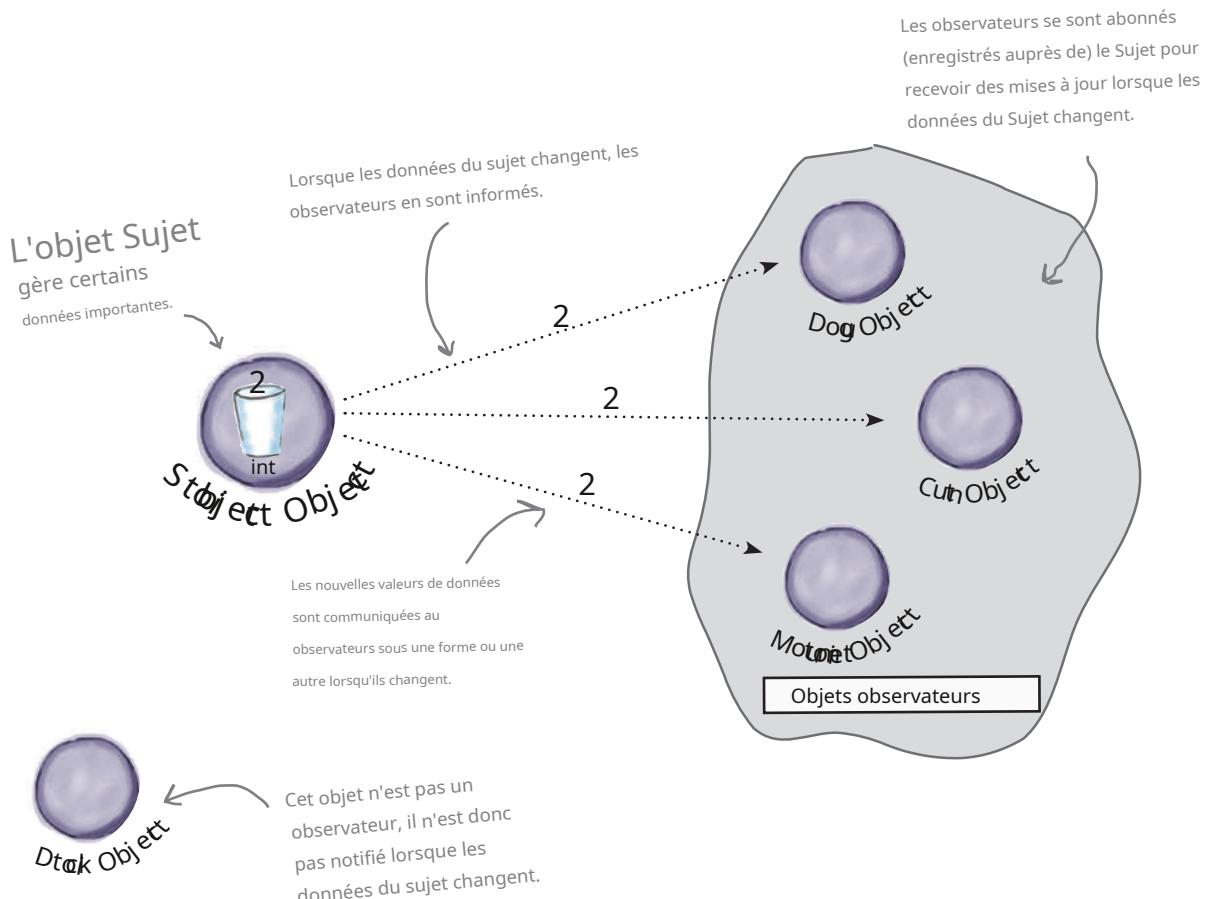
- 1** Un éditeur de journaux se lance en affaires et commence à publier des journaux.
- 2** Vous vous abonnez à un éditeur particulier et chaque fois qu'un nouveau numéro paraît, il vous est livré. Tant que vous restez abonné, vous recevez de nouveaux journaux.
- 3** Vous vous désabonnez lorsque vous ne voulez plus de journaux, et ils cessent d'être livrés.
- 4** Tant que l'éditeur est en activité, des particuliers, des hôtels, des compagnies aériennes et d'autres entreprises s'abonnent et se désabonnent constamment du journal.



Éditeurs + abonnés = modèle d'observateur

Si vous comprenez les abonnements aux journaux, vous comprenez à peu près le modèle d'Observer, sauf que nous appelons l'éditeur le **SUJET** et les abonnés les **OBSERVATEURS**.

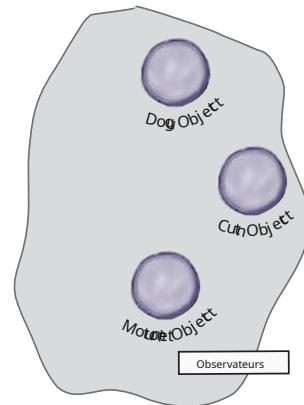
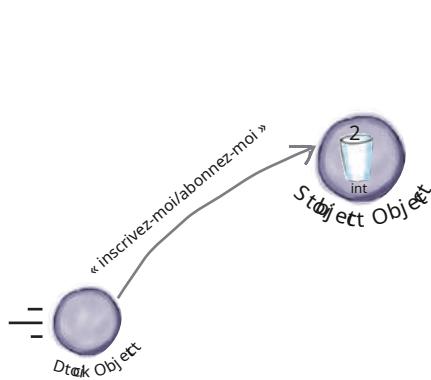
Regardons cela de plus près :



Une journée dans la vie du modèle Observer

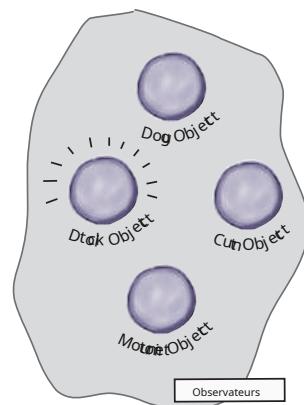
Un objet Canard arrive et dit au Sujet qu'il veut devenir observateur.

Duck veut vraiment participer à l'action ; ces ints que le sujet envoie à chaque changement d'état semblent assez intéressants...



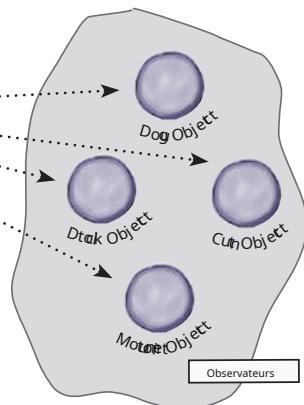
L'objet Canard est désormais un observateur officiel.

Duck est excité... il est sur la liste et attend avec impatience la prochaine notification pour pouvoir obtenir un int.



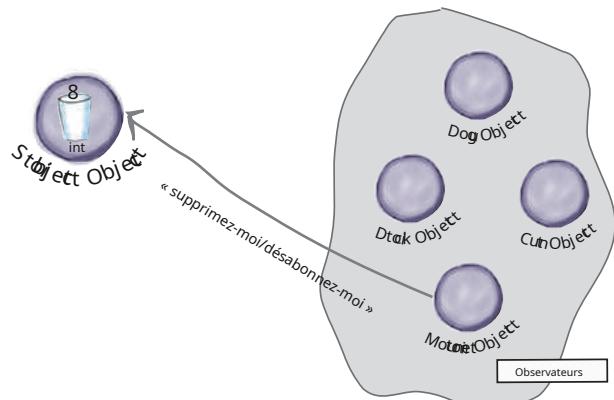
Le sujet obtient une nouvelle valeur de données !

Maintenant, Duck et tous les autres observateurs reçoivent une notification indiquant que le sujet a changé.



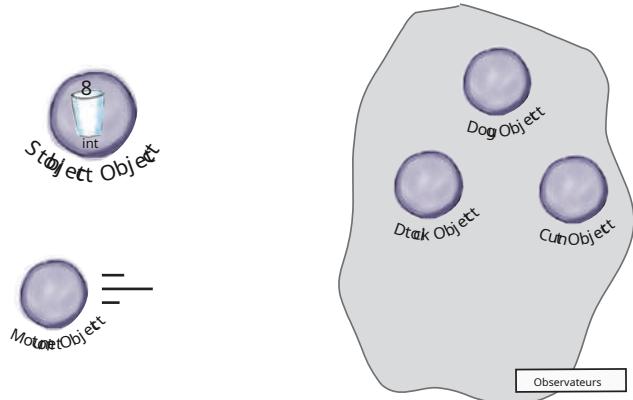
L'objet Souris demande à être supprimé en tant qu'observateur.

L'objet Souris reçoit des entrées depuis des lustres et en a assez, il décide donc qu'il est temps d'arrêter d'être un observateur.



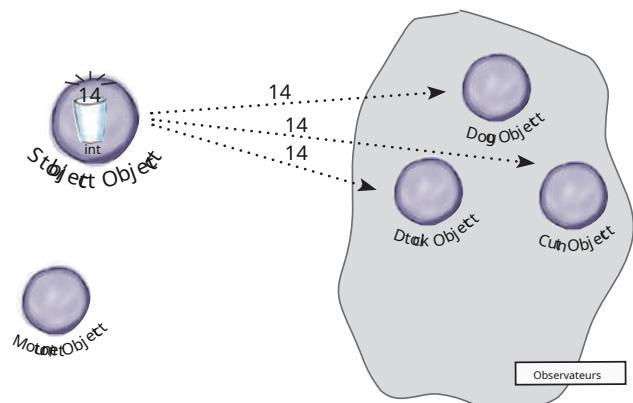
La souris est partie !

Le sujet reconnaît la demande de la souris et la retire de l'ensemble des observateurs.



Le sujet a un autre nouvel int.

Tous les observateurs reçoivent une nouvelle notification, à l'exception de la Souris qui n'est plus incluse. Ne le dites à personne, mais la Souris regrette secrètement ces ints... peut-être demandera-t-elle à redevenir observateur un jour.





Le drame de cinq minutes : un sujet d'observation

Dans le sketch d'aujourd'hui, deux développeurs de logiciels entreprenants rencontrent un véritable chasseur de têtes...

C'est Lori. Je recherche un poste de développement Java.
J'ai cinq ans d'expérience
d'expérience et...



1

Euh, ouais, toi et tous
les autres, bébé.

Je vous mets sur ma liste de
développeurs Java. Ne mappelez pas
moi je t'appelle !



2

Chasseur de têtes/Sujet

**Logiciel
Développeur #1**

Bonjour, je m'appelle Jill. J'ai écrit de
nombreux systèmes d'entreprise. Je suis
intéressée par tout travail que vous pourriez
avoir avec le développement Java.



3

**Logiciel
Développeur #2**

Je t'ajouterai à la liste,
tu le sauras avec tout le
monde.



4

Sujet

5 Pendant ce temps, la vie de Lori et Jill continue. Si un travail Java se présente, elles seront prévenues. Après tout, elles sont des observatrices.



6
Sujet

Jill décroche son propre emploi !



8
Observateur



7

Observateur

Observateur



9
Sujet

Deux semaines plus tard...



Jill adore la vie et n'est plus une observatrice. Elle profite également de la belle prime à la signature qu'elle a reçue parce que l'entreprise n'a pas eu à payer de chasseur de têtes.

Mais qu'est-il advenu de notre chère Lori ? On dit qu'elle a battu le chasseur de têtes à son propre jeu. Non seulement elle est toujours observatrice, mais elle a désormais sa propre liste d'appels et elle informe ses propres observateurs. Lori est à la fois sujet et observatrice.



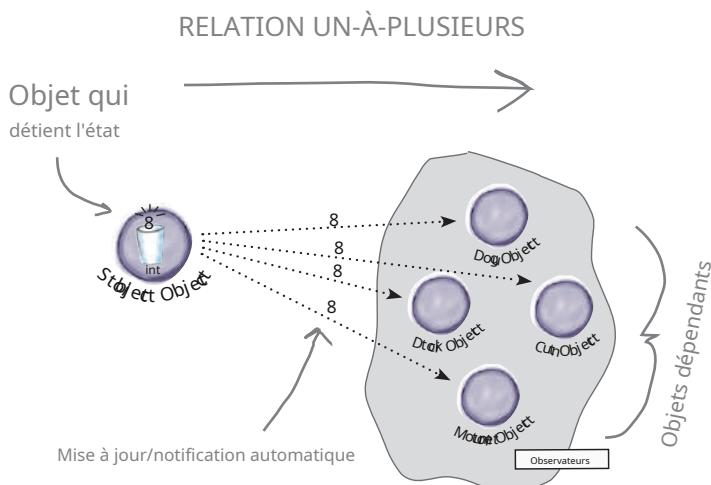
Définition du modèle d'observateur

Un abonnement à un journal, avec son éditeur et ses abonnés, est un bon moyen de visualiser le modèle.

Dans le monde réel, cependant, vous verrez généralement le modèle d'observateur défini comme ceci :

Le modèle d'observateur définit une dépendance un-à-plusieurs entre les objets de sorte que lorsqu'un objet change d'état, tous ses dépendants sont notifiés et mis à jour automatiquement.

Relions cette définition à la façon dont nous avons réfléchi au modèle :



Le modèle d'observateur définit un **un-à-plusieurs relation entre un ensemble d'objets**.

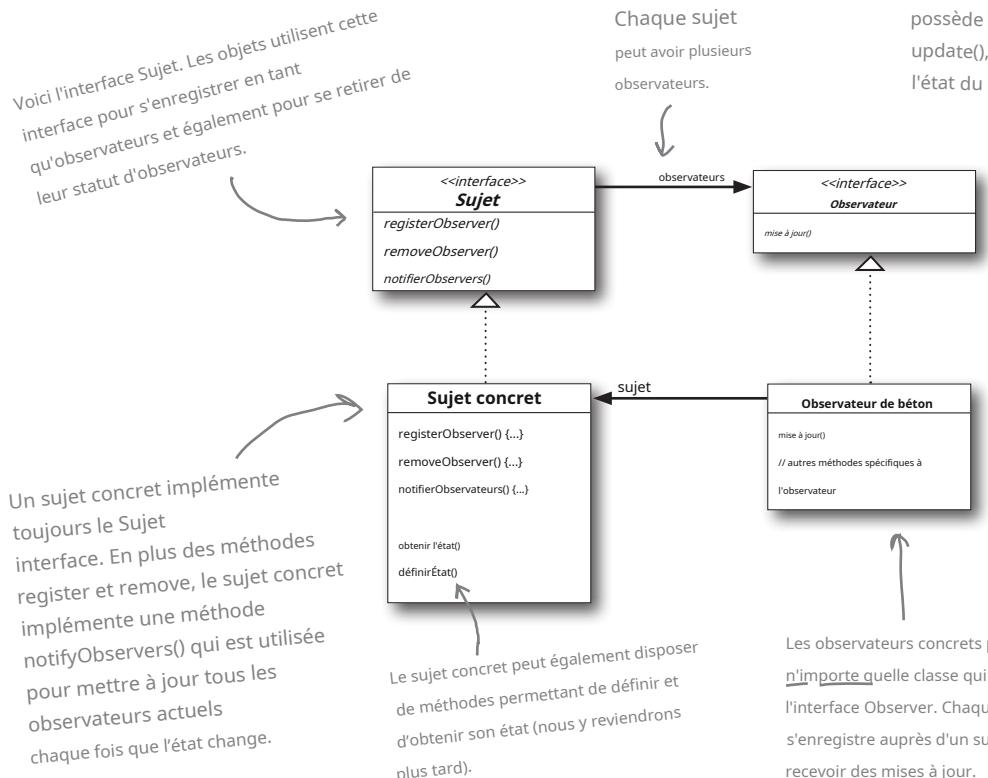
Lorsque l'état d'un objet change, tous ses dépendants sont notifiés.

Le sujet et les observateurs définissent la relation un-à-plusieurs. Nous avons *un sujet*, qui notifie *de nombreux observateurs* lorsque quelque chose dans le sujet change. Les observateurs sont *dépendants* sur le sujet : lorsque l'état du sujet change, les observateurs sont avertis.

Comme vous le découvrirez, il existe plusieurs façons différentes d'implémenter le modèle Observer, mais la plupart s'articulent autour d'une conception de classe qui inclut des interfaces *Subject* et *Observer*.

Le modèle d'observateur : le diagramme de classes

Examinons la structure du modèle Observer, avec ses classes Subject et Observer. Voici le diagramme de classes :



*there are no
Dumb Questions*

Q: Quel est le rapport avec les relations un-à-plusieurs ?

UN: Avec le modèle Observateur, le Sujet est l'objet qui contient l'état et le contrôle. Il y a donc UN sujet avec un état. Les observateurs, en revanche, utilisent l'état, même s'ils n'en sont pas propriétaires. Il y a de nombreux observateurs, et ils comptent sur le Sujet pour les informer lorsque son état change. Il existe donc une relation entre le Sujet UNIQUE et les NOMBREUX Observateurs.

Q: Comment la dépendance apparaît-elle ?

UN: Étant donné que le sujet est le seul propriétaire de ces données, les observateurs dépendent du sujet pour les mettre à jour lorsque les données changent. Cela conduit à une conception OO plus propre que si plusieurs objets pouvaient contrôler les mêmes données.

Q: J'ai également entendu parler du modèle de publication-abonnement. Est-ce simplement un autre nom pour le modèle d'observation ?

UN: Non, bien qu'ils soient liés. Le modèle Publier-S'abonner est un modèle plus complexe qui permet aux abonnés d'exprimer leur intérêt pour différents types de messages et sépare davantage les éditeurs des abonnés. Il est souvent utilisé dans les systèmes middleware.



Gourou et étudiant...

Gourou:Avons-nous parlé de couplage lâche ? **Étudiant:**

Guru, je ne me souviens pas d'une telle discussion.

Gourou:Un panier à tissage serré est-il rigide ou flexible ?

Étudiant:Raide, gourou.

Gourou:Et les paniers rigides ou flexibles se déchirent-ils ou se cassent-ils moins facilement ?

Étudiant:Un panier flexible a tendance à se casser moins facilement.

Gourou:Et dans nos logiciels, nos conceptions pourraient-elles se briser moins facilement si nos objets étaient moins étroitement liés les uns aux autres ?

Étudiant:Guru, je vois la vérité. Mais que signifie pour les objets d'être moins étroitement liés ?

Gourou:Nous aimons l'appeler « faiblement couplé ».

Étudiant:Ah!

Gourou:On dit qu'un objet est étroitement couplé à un autre objet lorsqu'il est aussi dépendant de cet objet.

Étudiant:Donc un objet faiblement couplé ne peut pas dépendre d'un autre objet ?

Gourou:Pensez à la nature : tous les êtres vivants dépendent les uns des autres. De même, tous les objets dépendent d'autres objets. Mais un objet faiblement couplé ne connaît pas ou ne se soucie pas vraiment des détails d'un autre objet.

Étudiant:Mais Guru, ça ne semble pas être une bonne qualité. Ne pas savoir est sûrement pire que savoir.

Gourou:Vous réussissez bien dans vos études, mais vous avez encore beaucoup à apprendre. En ne connaissant pas trop les autres objets, nous pouvons créer des modèles qui peuvent mieux gérer le changement. Des modèles plus flexibles, comme le panier à tissage moins serré.

Étudiant:Bien sûr, je suis sûr que vous avez raison. Pourriez-vous me donner un exemple ?

Gourou:Cela suffit pour aujourd'hui.

La puissance du couplage lâche

Lorsque deux objets sont *faiblement couplés*, ils peuvent interagir, mais ils ont généralement très peu de connaissances les uns des autres. Comme nous allons le voir, les conceptions faiblement couplées nous offrent souvent beaucoup de flexibilité (nous y reviendrons plus tard). Et, il s'avère que le modèle Observer est un excellent exemple de couplage faible. Examinons toutes les façons dont le modèle parvient à un couplage faible :

Premièrement, la seule chose que le sujet sait d'un observateur est qu'il implémente une certaine interface (l'interface Observer). Il n'a pas besoin de connaître la classe concrète de l'observateur, ce qu'il fait ou quoi que ce soit d'autre à son sujet.

Nous pouvons ajouter de nouveaux observateurs à tout moment. Comme la seule chose dont dépend le sujet est une liste d'objets qui implémentent l'interface Observer, nous pouvons ajouter de nouveaux observateurs quand nous le souhaitons. En fait, nous pouvons remplacer n'importe quel observateur au moment de l'exécution par un autre observateur et le sujet continuera à ronronner. De même, nous pouvons supprimer des observateurs à tout moment.

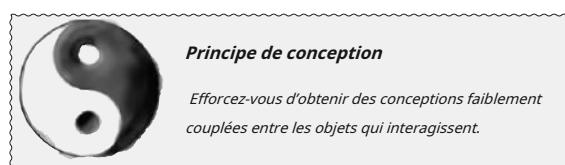
Nous n'avons jamais besoin de modifier le sujet pour ajouter de nouveaux types d'observateurs.

Supposons qu'une nouvelle classe concrète arrive et qu'elle doit être un observateur. Nous n'avons pas besoin d'apporter de modifications au sujet pour prendre en charge le nouveau type de classe ; tout ce que nous avons à faire est d'implémenter l'interface Observer dans la nouvelle classe et de nous inscrire en tant qu'observateur. Le sujet s'en fiche ; il enverra des notifications à tout objet qui implémente l'interface Observer.



Nous pouvons réutiliser des sujets ou des observateurs indépendamment les uns des autres. Si nous avons une autre utilisation d'un sujet ou d'un observateur, nous pouvons facilement les réutiliser car les deux ne sont pas étroitement liés.

Les modifications apportées au sujet ou à un observateur n'affecteront pas l'autre. Étant donné que les deux sont faiblement couplés, nous sommes libres d'apporter des modifications à l'un ou l'autre, tant que les objets répondent toujours à leurs obligations d'implémenter les interfaces Subject ou Observer.



Regardez ! Nous avons un nouveau principe de conception !

Les conceptions faiblement couplées nous permettent de créer des systèmes OO flexibles capables de gérer les changements, car ils minimisent l'interdépendance entre les objets.



Sharpen your pencil

Avant de continuer, essayez d'esquisser les classes dont vous aurez besoin pour implémenter la station météo, y compris la classe WeatherData et ses éléments d'affichage. Assurez-vous que votre diagramme montre comment toutes les pièces s'assemblent et également comment un autre développeur pourrait implémenter son propre élément d'affichage.

Si vous avez besoin d'un peu d'aide, lisez la page suivante ; vos coéquipiers discutent déjà de la façon de concevoir la station météo.

Conversation en cabine

Revenons au projet de la station météo. Vos coéquipiers ont déjà commencé à réfléchir au problème...



Marie:Eh bien, cela aide de savoir que nous utilisons le modèle Observer.

Poursuivre en justice:D'accord... mais comment l'appliquer ?

Marie:Hmm. Regardons à nouveau la définition :

Le modèle Observer définit une dépendance un-à-plusieurs entre les objets afin que lorsqu'un objet change d'état, tous ses dépendants soient notifiés et mis à jour automatiquement.

Marie:En fait, cela a du sens quand on y pense. Notre classe WeatherData est la « une » et notre « plusieurs » correspond aux différents éléments d'affichage qui utilisent les mesures météorologiques.

Poursuivre en justice:C'est vrai. La classe WeatherData a certainement un état... c'est-à-dire la température, l'humidité et la pression barométrique, et ces éléments changent certainement.

Marie:Oui, et lorsque ces mesures changent, nous devons avertir tous les éléments d'affichage afin qu'ils puissent faire ce qu'ils veulent avec les mesures.

Poursuivre en justice:Cool, maintenant je pense que je vois comment le modèle d'observateur peut être appliqué à notre problème de station météo.

Marie:Il y a encore quelques points à prendre en compte que je ne suis pas sûr de comprendre encore. **Poursuivre en justice:**Comme quoi?

Marie:D'une part, comment pouvons-nous transmettre les mesures météorologiques aux éléments d'affichage ?

Poursuivre en justice:Eh bien, en revenant à l'image du modèle Observer, si nous faisons de l'objet WeatherData le sujet et des éléments d'affichage les observateurs, alors les écrans s'enregistreront auprès de l'objet WeatherData afin d'obtenir les informations qu'ils souhaitent, n'est-ce pas ?

Marie:Oui... et une fois que la station météo connaît un élément d'affichage, elle peut simplement appeler une méthode pour l'informer des mesures.

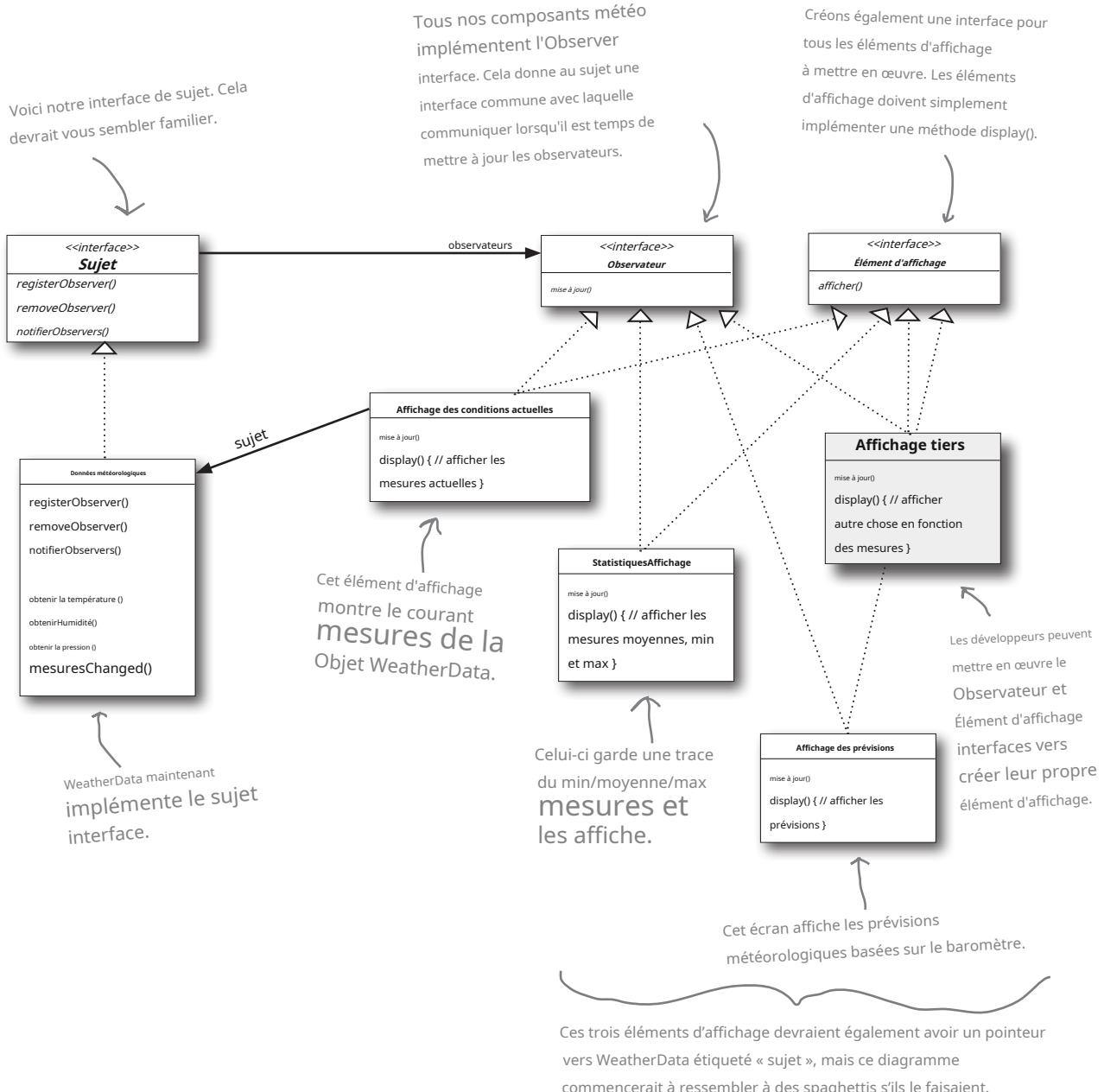
Poursuivre en justice:Il ne faut pas oublier que chaque élément d'affichage peut être différent... c'est pourquoi je pense que c'est là qu'intervient une interface commune. Même si chaque composant a un type différent, ils doivent tous implémenter la même interface afin que l'objet WeatherData sache comment leur envoyer les mesures.

Marie:Je vois ce que tu veux dire. Ainsi, chaque écran aura, par exemple, une méthode update() que WeatherData appellera.

Poursuivre en justice:Et update() est défini dans une interface commune que tous les éléments implémentent...

Conception de la station météo

Comment ce diagramme se compare-t-il au vôtre ?



Mise en place de la station météo

D'accord, nous avons eu de bonnes idées de la part de Mary et Sue (depuis quelques pages) et nous avons un diagramme qui détaille la structure globale de nos classes. Alors, commençons notre implémentation de la station météo. Commençons par les interfaces :

interface publique Sujet {

```
    public void registerObserver(Observateur o); public  
    void removeObserver(Observateur o); public void  
    notifyObservers();
```

}

Ces deux méthodes prennent un observateur comme argument, c'est-à-dire l'observateur à enregistrer ou à supprimer.

Cette méthode est appelée pour notifier tous les observateurs lorsque l'état du sujet a changé.

interface publique Observateur {

```
    public void update(température flottante, humidité flottante, pression flottante);
```

}

Ce sont les valeurs d'état que les observateurs obtiennent du sujet lorsqu'une mesure météorologique change.

interface publique DisplayElement {

```
    public void affichage();
```

}



L'interface DisplayElement inclut simplement une méthode, display(), que nous appellerons lorsque l'élément d'affichage doit être affiché.

L'interface de l'observateur est implémenté par tous les observateurs, ils doivent donc tous implémenter la méthode update(). Ici nous suivons l'exemple de Mary et Sue et transmettons les mesures aux observateurs.



Mary et Sue pensaient que transmettre les mesures directement aux observateurs était la méthode la plus simple pour mettre à jour l'état. Pensez-vous que cela soit judicieux ? Indice : s'agit-il d'un domaine de l'application qui pourrait changer à l'avenir ? Si cela devait changer, le changement serait-il bien encapsulé ou nécessiterait-il des modifications dans de nombreuses parties du code ?

Pouvez-vous penser à d'autres façons d'aborder le problème de la transmission de l'état mis à jour aux observateurs ?

Ne vous inquiétez pas, nous reviendrons sur cette décision de conception après avoir terminé la mise en œuvre initiale.

Implémentation de l'interface Subject dans WeatherData

Vous vous souvenez de notre première tentative d'implémentation de la classe WeatherData au début du chapitre ? Vous devriez peut-être vous rafraîchir la mémoire. Il est maintenant temps de revenir en arrière et de faire les choses en gardant à l'esprit le modèle Observer :

```

la classe publique WeatherData implémente Subject {
    Liste privée <Observer> observateurs ;
    température flottante privée ;
    humidité du flotteur privé;
    pression du flotteur privé;

    données météorologiques publiques () {
        observateurs = new ArrayList<Observateur>();
    }

    public void registerObserver(Observateur o) {
        observateurs.add(o);
    }

    public void removeObserver(Observateur o) {
        observateurs.remove(o);
    }

    public void notifierObservers() {
        pour (Observateur observateur : observateurs) {
            observateur.update(température, humidité, pression);
        }
    }

    public void mesuresChanged() {
        notifierObservers();
    }

    public void setMeasurements(float température, float humidité, float pression) {
        this.temperature = température;
        this.humidity = humidité; this.pressure
        = pression; measurementsChanged();
    }

    // autres méthodes WeatherData ici
}

```

Ici, nous implémentons l'interface Subject.

N'OUBLIEZ PAS : nous ne fournissons pas d'instructions d'importation et de package dans les listes de codes.

Obtenez le code source complet à partir de <https://wickedlysmart.com/head-first-design-patterns>

WeatherData implémente désormais l'interface Subject.

Nous avons ajouté une ArrayList pour contenir les observateurs, et nous la créons dans le constructeur.

Lorsqu'un observateur s'inscrit, nous l'ajoutons simplement à la fin de la liste.

De même, lorsqu'un observateur souhaite se désinscrire, nous le retirons simplement de la liste.

Voici la partie amusante : c'est là que nous informons tous les observateurs de l'état. Comme ils sont tous des observateurs, nous savons qu'ils implémentent tous update(), nous savons donc comment les avertir.

Nous informons les observateurs lorsque nous recevons des mesures mises à jour de la station météo.

Bon, nous voulions bien fournir une jolie petite station météo avec chaque livre, mais l'éditeur n'a pas voulu. Donc, plutôt que de lire les données météorologiques réelles sur un appareil, nous allons utiliser cette méthode pour tester nos éléments d'affichage. Ou, pour le plaisir, vous pouvez écrire du code pour récupérer des mesures sur le Web.

Maintenant, construisons ces éléments d'affichage

Maintenant que nous avons mis au point notre classe WeatherData, il est temps de construire les éléments d'affichage. Weather-O-Rama en a commandé trois : l'affichage des conditions actuelles, l'affichage des statistiques et l'affichage des prévisions. Jetons un œil à l'affichage des conditions actuelles ; une fois que vous aurez une bonne idée de cet élément d'affichage, consultez les affichages des statistiques et des prévisions dans le répertoire de code. Vous verrez qu'ils sont très similaires.

```
Cet affichage implémente l'interface  
Observer afin de pouvoir obtenir les  
modifications de l'objet WeatherData.  
  
Il implémente également  
DisplayElement, car notre API va  
exiger que tous les éléments d'affichage  
implémentent cette interface.  
  
la classe publique CurrentConditionsDisplay implémente Observer, DisplayElement {  
    température flottante privée ; humidité  
    flottante privée ; données météorologiques  
    privées WeatherData ;  
  
    public CurrentConditionsDisplay(DonnéesMétéoDonnéesMétéo) {  
        ceci.weatherData = donnéesmétéo;  
        donnéesmétéo.registerObserver(ceci);  
    }  
  
    public void update(température flottante, humidité flottante, pression flottante) {  
        this.temperature = température;  
        this.humidity = humidité; affichage();  
    }  
  
    affichage public void() {  
        System.out.println("Conditions actuelles : " + température  
            + "degrés F et " + humidité + "% d'humidité");  
    }  
}
```

Le constructeur reçoit l'objet weatherData (le sujet) et nous l'utilisons pour enregistrer l'affichage en tant qu'observateur.

Lorsque update() est appelé, nous enregistrons la température et l'humidité et appelons display().

La méthode display() imprime simplement la température et l'humidité les plus récentes.

there are no
Dumb Questions

Q: update() est-il le meilleur endroit pour appeler display() ?

UN: Dans cet exemple simple, il était logique d'appeler display() lorsque les valeurs changeaient. Cependant, vous avez raison ; il existe de bien meilleures façons de concevoir la manière dont les données sont affichées. Nous verrons cela lorsque nous aborderons le modèle Modèle-Vue-Contrôleur.

Q: Pourquoi avez-vous stocké une référence au sujet WeatherData ? Il ne semble pas que vous l'utilisiez à nouveau après le constructeur.

UN: C'est vrai, mais à l'avenir, nous voudrons peut-être nous désinscrire en tant qu'observateur et il serait pratique d'avoir déjà une référence au sujet.

Allumez la station météo



1 Commençons par créer un harnais de test.

La station météo est prête à fonctionner. Il ne nous manque plus qu'un peu de code pour tout assembler. Nous ajouterons quelques écrans supplémentaires et généraliserons les choses dans un instant. Pour l'instant, voici notre première tentative :

```
classe publique WeatherStation {  
  
    public static void main(String[] args) {  
        DonnéesMétéo donnéesMétéo = new DonnéesMétéo();  
  
        Affichage des conditions actuelles Affichage actuel =  
            nouveau CurrentConditionsDisplay(weatherData);  
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData); ForecastDisplay  
        forecastDisplay = new ForecastDisplay(weatherData);  
  
        weatherData.setMeasurements(80, 65, 30.4f);  
        weatherData.setMeasurements(82, 70, 29.2f);  
        weatherData.setMeasurements(78, 90, 29.2f);  
    }  
}
```

Si vous ne le faites pas
vouloir
télécharger le
code, tu peux
commenter
ces deux lignes
et exécutez-le.

Tout d'abord, créez le
Objet WeatherData.

Simuler une nouvelle météo
mesures.

Créez les trois
affiche et
passez-leur le
Objet WeatherData.

2 Exécutez le code et laissez le modèle Observer faire sa magie.

```
Aide de la fenêtre d'édition de fichier StormyWeather  
  
Station météo Java  
Conditions actuelles : 80,0 °F et 65,0 % d'humidité. Température  
moyenne/max./min. = 80,0/80,0/80,0  
Prévisions : Amélioration de la météo en route !  
Conditions actuelles : 82,0 °F et 70,0 % d'humidité. Température  
moyenne/max./min. = 81,0/82,0/80,0  
Prévisions : Attention au temps plus frais et pluvieux Conditions  
actuelles : 78,0°F et 90,0 % d'humidité Température moyenne/  
max./min. = 80,0/82,0/78,0  
Prévisions : Toujours la même chose
```



Johnny Hurricane, le PDG de Weather-O-Rama, vient d'appeler et il est impossible qu'ils livrent sans un élément d'affichage de l'indice de chaleur. Voici les détails.

L'indice de chaleur est un indice qui combine la température et l'humidité pour déterminer la température apparente (la chaleur ressentie). Pour calculer l'indice de chaleur, vous prenez la température, T, et l'humidité relative, RH, et vous utilisez cette formule :

indice de chaleur =

$$\begin{aligned}
 & 16.923 + 1,85212 * 10^{-1} * T + 5,37941 * RH - 1,00254 * 10^{-1} * \\
 & T * RH + 9,41695 * 10^{-3} * T_2 + 7,28898 * 10^{-3} * RH_2 + 3,45372 * \\
 & 10^{-4} * T_2 * RH - 8,14971 * 10^{-4} * T * RH_2 + 1,02102 * 10^{-5} * T_2 * \\
 & RH_2 - 3,8646 * 10^{-5} * T_3 + 2,91583 * 10^{-5} * RH_3 + 1,42721 * 10^{-6} \\
 & * T_3 * HR + 1,97483 * 10^{-7} * T * RH_3 - 2,18429 * 10^{-8} * T_3 * RH_2 \\
 & + 8,43296 * 10^{-10} * T_2 * RH_3 - 4,81975 * 10^{-11} * T_3 * RH_3
 \end{aligned}$$

Alors, commencez à écrire !

Je plaisante. Ne vous inquiétez pas, vous n'aurez pas à saisir cette formule ; créez simplement la vôtre *Affichage de l'indice de chaleur.java* fichier et copiez la formule à partir de *indice de chaleur.txt* dedans.



Vous pouvez obtenir *heatindex.txt* sur wickedlysmart.com.

Comment ça marche ? Il faudrait se référer à *Météorologie de la tête la première*, ou essayez de demander à quelqu'un du National Weather Service (ou essayez une recherche sur le Web).

Une fois terminé, votre résultat devrait ressembler à ceci :

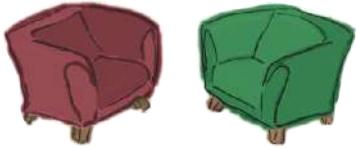
Voici ce que
changé en
cette sortie.

```

Aide de la fenêtre d'édition de fichier OverDaRainbow
Station météo Java
Conditions actuelles : 80,0 °F et 65,0 % d'humidité.
Température moyenne/max./min. = 80,0/80,0/80,0
Prévisions : amélioration des conditions météorologiques !
L'indice de chaleur est de 82,95535
Conditions actuelles : 82,0 °F et 70,0 % d'humidité.
Température moyenne/max./min. = 81,0/82,0/80,0
Prévisions : Attention au temps plus frais et pluvieux L'indice
de chaleur est de 86,90124
Conditions actuelles : 78,0 °F et 90,0 % d'humidité.
Température moyenne/max./min. = 80,0/82,0/78,0
Prévisions : La même chose se produira
L'indice de chaleur est de 83,64967 %

```

Fireside Chats



La conférence de ce soir : Le sujet et l'observateur se disputent sur la bonne façon de transmettre les informations d'état à l'observateur.

Sujet:

Je suis heureux que nous ayons enfin la chance de discuter en personne.

Observateur:

Vraiment ? Je pensais que vous ne vous souciez pas beaucoup de nous, les observateurs.

Eh bien, je fais mon travail, n'est-ce pas ? Je vous dis toujours ce qui se passe... Ce n'est pas parce que je ne sais pas vraiment qui vous êtes que je m'en fiche. Et puis, je sais la chose la plus importante à votre sujet : vous implémentez l'interface Observer.

Ouais, mais ce n'est qu'une petite partie de qui je suis. Quoi qu'il en soit, j'en sais beaucoup plus sur toi...

Oh ouais, comme quoi ?

Eh bien, vous nous transmettez toujours votre état, à nous les observateurs, afin que nous puissions voir ce qui se passe à l'intérieur de vous. Ce qui devient un peu ennuyeux parfois...

Eh bien, excusez-moi. Je dois envoyer mon état avec mes notifications pour que tous les observateurs paresseux sachent ce qui s'est passé !

Ok, attendez une minute ici ; premièrement, nous ne sommes pas paresseux, nous avons juste d'autres choses à faire entre vos notifications si importantes, Monsieur le Sujet, et deuxièmement, pourquoi ne nous laissez-vous pas venir à vous pour l'état que nous voulons plutôt que de le proposer à tout le monde ?

Eh bien... je suppose que ça pourrait marcher. Mais je devrais m'ouvrir encore plus pour laisser tous les observateurs entrer et obtenir l'état dont vous avez besoin. Cela pourrait être un peu dangereux. Je ne peux pas vous laisser entrer et fouiner dans tout ce que j'ai.

Sujet:

Oui, je pourrais te laisser **tirer** mon état. Mais cela ne sera-t-il pas moins pratique pour vous ? Si vous devez venir me voir à chaque fois que vous voulez quelque chose, vous devrez peut-être effectuer plusieurs appels de méthode pour obtenir tout l'état que vous souhaitez. C'est pourquoi j'aime **pousser** mieux...alors vous avez tout ce dont vous avez besoin dans une seule notification.

Observateur:

Pourquoi n'écrivez-vous pas simplement quelques méthodes getter publiques qui nous permettront d'extraire l'état dont nous avons besoin ?

Ne soyez pas si insistant ! Il existe tellement de types d'observateurs différents qu'il est impossible d'anticiper tout ce dont nous avons besoin. Laissez-nous simplement venir à vous pour obtenir l'état dont nous avons besoin. De cette façon, si certains d'entre nous n'ont besoin que d'un petit peu d'état, nous ne sommes pas obligés de tout obtenir. Cela facilite également les modifications ultérieures. Disons, par exemple, que vous nous développez et ajoutez un peu plus d'état. Si vous utilisez pull, vous n'avez pas besoin de faire le tour et de modifier les appels de mise à jour sur chaque observateur ; vous devez simplement modifier pour permettre à davantage de méthodes getter d'accéder à notre état supplémentaire.

Eh bien, comme j'aime le dire, ne nous appelez pas, nous vous appellerons ! Mais je vais y réfléchir.

Je ne retiendrai pas mon souffle.

On ne sait jamais, bon sang *pourrait* geler.

Je vois, toujours le gars sage...

En effet.

À la recherche du modèle d'observateur dans la nature

Le modèle d'observateur est l'un des modèles les plus courants et vous trouverez de nombreux exemples de ce modèle utilisé dans de nombreuses bibliothèques et frameworks. Si nous examinons le Java Development Kit (JDK), par exemple, les bibliothèques JavaBeans et Swing utilisent toutes deux le modèle Observer. Le modèle ne se limite pas non plus à Java ; il est utilisé dans les événements de JavaScript et dans le protocole Key-Value Observing de Cocoa et Swift, pour ne citer que quelques autres exemples. L'un des avantages de connaître les modèles de conception est de reconnaître et de comprendre rapidement la motivation de conception dans vos bibliothèques préférées. Faisons un rapide détour par la bibliothèque Swing pour voir comment Observer est utilisé.



Si vous êtes curieux à propos du modèle d'observateur dans JavaBeans, consultez le `PropertyChangeListener` interface.

La bibliothèque Swing

Vous savez probablement déjà que Swing est la boîte à outils de l'interface utilisateur graphique de Java. L'un des composants les plus basiques de cette boîte à outils est la classe JButton. Si vous recherchez la superclasse de JButton, AbstractButton, vous constaterez qu'elle possède de nombreuses méthodes d'ajout/suppression d'écouteurs. Ces méthodes vous permettent d'ajouter et de supprimer des observateurs : ou, comme on les appelle dans Swing, des écouteurs, pour écouter différents types d'événements qui se produisent sur le composant Swing. Par exemple, un ActionListener vous permet d'« écouter » tout type d'action qui peut se produire sur un bouton, comme une pression sur un bouton. Vous trouverez différents types d'écouteurs dans toute l'API Swing.

Une petite application qui change la vie

D'accord, notre application est assez simple. Vous avez un bouton qui dit : « Dois-je le faire ? » et lorsque vous cliquez sur ce bouton, les auditeurs (observateurs) peuvent répondre à la question comme ils le souhaitent. Nous implémentons deux de ces auditeurs, appelés AngelListener et DevilListener. Voici comment l'application se comporte :

Voici notre interface sophistiquée.

Should I do it?

Réponse du diable

Réponse de l'ange

Fenêtre d'édition de fichier Aide HeMadeMeDoIt

```
%java SwingObserverExample
Allez, fais-le !
Ne le faites pas, vous pourriez le regretter ! %
```

Coder l'application qui va changer la vie

Cette application révolutionnaire nécessite très peu de code. Il suffit de créer un objet JButton, de l'ajouter à un JFrame et de configurer nos écouteurs. Nous allons utiliser des classes internes pour les écouteurs, ce qui est une technique courante dans la programmation Swing. Si vous n'êtes pas au courant des classes internes ou de Swing, vous devriez peut-être consulter le chapitre Swing de votre guide de référence Java préféré.

```
classe publique SwingObserverExample {  
    Cadre JFrame;  
    public static void main(String[] args) {  
        SwingObserverExample exemple = new SwingObserverExample();  
        exemple.go();  
    }  
    public void go() {  
        cadre = nouveau JFrame();  
  
        JButton button = new JButton("Dois-je le faire ?");  
        button.addActionListener(new AngelListener());  
        button.addActionListener(new DevilListener());  
  
        // Définir les propriétés du cadre ici  
    }  
  
    la classe AngelListener implémente ActionListener {  
        public void actionPerformed(événement ActionEvent) {  
            System.out.println("Ne le faites pas, vous pourriez le regretter!");  
        }  
    }  
  
    la classe DevilListener implémente ActionListener {  
        public void actionPerformed(événement ActionEvent) {  
            System.out.println("Allez, fais-le!");  
        }  
    }  
}
```

Application Swing simple qui crée simplement un cadre et y place un bouton.

Fait des objets diable et ange des auditeurs (observateurs) du bouton.

← Le code pour configurer le cadre va ici.

← Voici les définitions de classe pour les observateurs, définies comme des classes internes (mais ce n'est pas obligatoire).

← Plutôt que update(), la méthode actionPerformed() est appelée lorsque l'état du sujet (dans ce cas, le bouton) change.



Serious Coding

Les expressions lambda ont été ajoutées dans Java 8. Si vous ne les connaissez pas, ne vous inquiétez pas ; vous pouvez continuer à utiliser des classes internes pour vos observateurs Swing.



Et si vous pouviez encore plus loin votre utilisation du modèle Observer ? En utilisant une expression lambda plutôt qu'une classe interne, vous pouvez ignorer l'étape de création d'un objet ActionListener. Avec une expression lambda, nous créons à la place un objet fonction, et l'objet fonction est l'observateur. Et, lorsque vous transmettez cet objet de fonction à addActionListener(), Java garantit que sa signature correspond à actionPerformed(), la seule méthode de l'interface ActionListener.

Plus tard, lorsque le bouton est cliqué, l'objet bouton informe ses observateurs, y compris les objets de fonction créés par les expressions lambda, qu'il a été cliqué et appelle la méthode actionPerformed() de chaque auditeur.

Voyons comment vous utiliseriez les expressions lambda comme observateurs pour simplifier notre code précédent :

Le code mis à jour, utilisant des expressions lambda :

```
classe publique SwingObserverExample {
    Cadre JFrame;
    public static void main(String[] args) {
        SwingObserverExample exemple = new SwingObserverExample();
        exemple.go();
    }
    public void go() {
        cadre = nouveau JFrame();

        JButton button = new JButton("Dois-je le faire ?");
        button.addActionListener(event ->
            System.out.println("Ne le faites pas, vous pourriez le regretter !"));
        button.addActionListener(event ->
            System.out.println("Allez, fais-le !"));

        // Définir les propriétés du cadre ici
    }
}
```

Nous avons remplacé les objets AngelListener et DevilListener par des expressions lambda qui implémentent les mêmes fonctionnalités que celles que nous avions auparavant.

Lorsque vous cliquez sur le bouton, les objets de fonction créés par les expressions lambda sont notifiés et la méthode qu'ils implémentent est exécutée.

L'utilisation d'expressions lambda rend ce code beaucoup plus concis.

Pour en savoir plus sur les expressions lambda, consultez la documentation Java.

there are no Dumb Questions

Q: Je pensais que Java avait des classes `Observer` et `Observable` ?

UN: Bonne question. Java fournissait auparavant une classe `Observable` (le sujet) et une interface `Observer`, que vous pouviez utiliser pour aider à intégrer le modèle `Observer` dans votre code. La classe `Observable` fournissait des méthodes pour ajouter, supprimer et notifier les observateurs, de sorte que vous n'aviez pas à écrire ce code. Et l'interface `Observer` fournissait une interface comme la nôtre, avec une méthode `update()`. Ces classes ont été déconseillées dans Java 9. Les gens trouvent plus facile de prendre en charge le modèle `Observer` de base dans leur propre code, ou veulent quelque chose de plus robuste, donc les classes `Observer/Observable` sont progressivement supprimées.

Q: Java propose-t-il un autre support intégré pour `Observer` pour remplacer ces classes ?

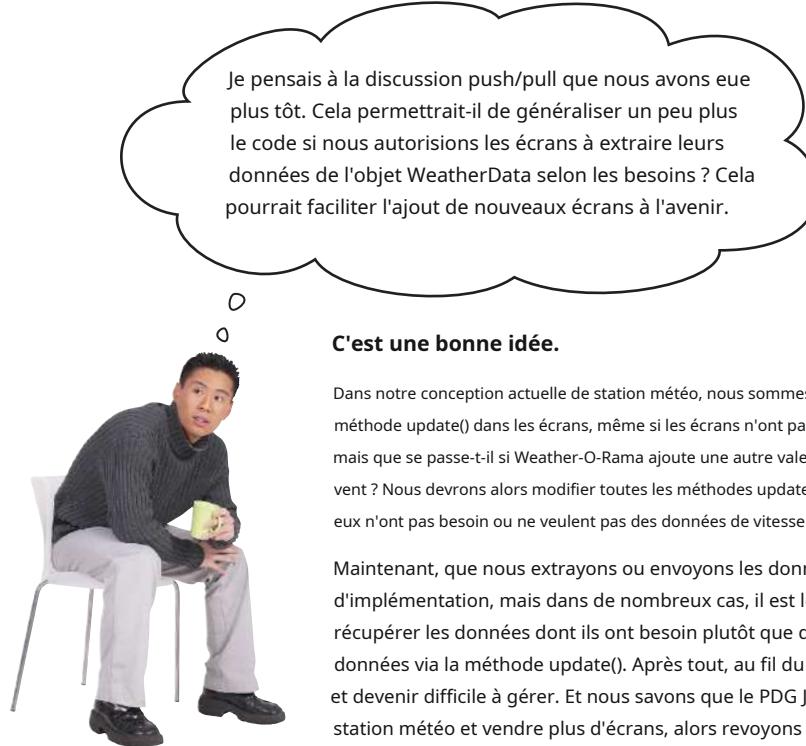
UN:

JavaBeans offre un support intégré via les `PropertyChangeEvents` qui sont générés lorsqu'un Bean modifie un type particulier de propriété et envoie des notifications aux `PropertyChangeListener`s. Il existe également des composants éditeur/abonné associés dans l'API Flow pour la gestion des flux asynchrones.

Q: Dois-je m'attendre à ce que les notifications d'un sujet à ses observateurs arrivent dans un ordre spécifique ?

UN:

Avec les implémentations Java d'`Observer`, les développeurs du JDK vous conseillent spécifiquement de ne pas dépendre d'un ordre de notification spécifique.



Dans notre conception actuelle de station météo, nous sommes *poussés* les trois éléments de données dans la méthode `update()` dans les écrans, même si les écrans n'ont pas besoin de toutes ces valeurs. Ce n'est pas grave, mais que se passe-t-il si Weather-O-Rama ajoute une autre valeur de données plus tard, comme la vitesse du vent ? Nous devrons alors modifier toutes les méthodes `update()` dans tous les écrans, même si la plupart d'entre eux n'ont pas besoin ou ne veulent pas des données de vitesse du vent.

Maintenant, que nous extrayons ou envoyons les données à l'`Observer` est un détail d'implémentation, mais dans de nombreux cas, il est logique de laisser les observateurs récupérer les données dont ils ont besoin plutôt que de leur transmettre de plus en plus de données via la méthode `update()`. Après tout, au fil du temps, c'est un domaine qui peut changer et devenir difficile à gérer. Et nous savons que le PDG Johnny Hurricane va vouloir agrandir la station météo et vendre plus d'écrans, alors revoyons la conception et voyons si nous pouvons la rendre encore plus facile à étendre à l'avenir.

Mise à jour du code de la station météo pour permettre aux observateurs de *tirer* les données dont ils ont besoin est un exercice assez simple. Tout ce que nous devons faire est de nous assurer que le sujet dispose de méthodes de récupération pour ses données, puis de modifier nos observateurs pour les utiliser afin d'extraire les données adaptées à leurs besoins. Faisons cela.



Pendant ce temps, de retour à Weather-O-Rama

Il existe une autre façon de gérer les données du sujet : nous pouvons compter sur les observateurs pour les extraire du sujet selon les besoins. À l'heure actuelle, lorsque les données du sujet changent, nous pousserons les nouvelles valeurs de température, d'humidité et de pression aux observateurs, en transmettant ces données dans l'appel à `update()`.

Configurons les choses de sorte que lorsqu'un observateur est informé d'un changement, il appelle les méthodes `getter` sur le sujet pour *tirer*/les valeurs dont elle a besoin.

Pour passer à l'utilisation de pull, nous devons apporter quelques petites modifications à notre code existant.

Pour que le sujet envoie des notifications...

- 1 Nous allons modifier la méthode `notifyObservers()` dans `WeatherData` pour appeler la méthode `update()` dans les Observateurs sans arguments :

```
public void notifierObservers() {
    pour (Observateur observateur : observateurs) {
        observateur.update();
    }
}
```

Pour qu'un observateur reçoive des notifications...

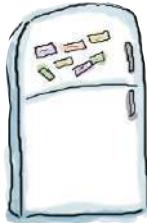
- 1 Ensuite, nous allons modifier l'interface `Observer`, en changeant la signature de la méthode `update()` afin qu'elle n'ait aucun paramètre :

```
interface publique Observateur {
    public void update();
}
```

- 2 Enfin, nous modifions chaque observateur concret pour changer la signature de son observateur respectif. Utilisez les méthodes `update()` et récupérez les données météo du sujet à l'aide des méthodes `getter` de `WeatherData`. Voici le nouveau code de la classe `CurrentConditionsDisplay` :

```
vide public mise à jour() {
    this.temperature = weatherData.getTemperature();
    this.humidity = weatherData.getHumidity(); display();
}
```

Ici, nous utilisons les méthodes `getter` du sujet qui ont été fournies avec le code dans `WeatherData` de Weather-O-Rama.



Codes aimantés

La classe ForecastDisplay est toute mélangée sur le réfrigérateur. Pouvez-vous reconstruire les extraits de code pour la faire fonctionner ? Certaines des accolades sont tombées par terre et elles étaient trop petites pour être ramassées, alors n'hésitez pas à en ajouter autant que vous le souhaitez !

public ForecastDisplay(Données Météo
données météo) {

afficher();

weatherData.registerObserver(ceci);

classe publique ForecastDisplay implémente
Observer, DisplayElement {

affichage public void() {
// afficher le code ici
}

dernière pression = pression actuelle; pression actuelle
= weatherData.getPressure();

flotteur privé currentPressure = 29,92f; flotteur
privé lastPressure;

ceci.weatherData = données météo;

public void mise à jour() {

}

Données météorologiques privées données météorologiques;

Testez le nouveau code



Ok, il vous reste un affichage à mettre à jour, l'affichage Moyenne/Min/Max. Allez-y, faites-le maintenant !

Juste pour être sûr, exécutons le nouveau code...

Voici ce que nous avons obtenu.



Regardez ! Cela vient d'arriver !



Fenêtre d'édition de fichier Aide TryThisAtHome

Station météo Java

Conditions actuelles : 80,0 °F et 65,0 % d'humidité. Température moyenne/max./min. = 80,0/80,0/80,0

Prévisions : Amélioration de la météo en route !

Conditions actuelles : 82,0 °F et 70,0 % d'humidité. Température moyenne/max./min. = 81,0/82,0/80,0

Prévisions : Attention au temps plus frais et pluvieux Conditions actuelles : 78,0°F et 90,0 % d'humidité Température moyenne/max./min. = 80,0/82,0/78,0

Prévisions : Toujours la même chose



Météo-O-Rama, Inc.
100, rue Main
Allée des tornades, OK 45021

Ouah!

Votre conception est fantastique. Non seulement vous avez rapidement créé les trois écrans que nous vous avions demandés, mais vous avez également créé une conception générale qui permet à quiconque de créer un nouvel écran et même aux utilisateurs d'ajouter et de supprimer des écrans au moment de l'exécution !

Ingénieux!

Jusqu'à notre prochain engagement,

Johnny Hurricane



Des outils pour votre boîte à outils de conception

Bienvenue à la fin du chapitre 2. Vous avez ajouté quelques nouveautés à votre boîte à outils OO...

Principes OO

Notions de base sur OO

- Abstraction
- Encapsulation
- Polymorphisme
- Héritage

Privilégier la composition plutôt que l'héritage.

Programmez vers des interfaces, pas vers des implémentations.

Efforcez-vous d'obtenir des conceptions faiblement couplées entre les objets qui interagissent.

Modèles OO

Stratégie - définit une famille d'algorithmes, encapsulebmme et un c.a.lhvouETHE, -andemfainkeess taheomne-à-beaucoup interchangandgepaebnled. enScyratbeegtywleetens tohbejeaclgtosritsohmthat varient indewpehnedenotnlye forbojmecctiecnhtas ntgheast suteatite., tous ses les personnes à charge sont notifiées et mises à jour automatiquement

Un nouveau modèle de communication état g à un ensemble d'objets de manière faiblement couplée. Nous n'avons pas vu le dernier mot du modèle Observer — attendez juste que nous parlions de MVC !



BULLET POINTS

- f Le modèle Observer définit une relation un-à-plusieurs entre les objets.
- f Les sujets mettent à jour les observateurs à l'aide d'une interface commune.
- f Les observateurs de tout type concret peuvent participer au modèle à condition qu'ils implémentent l'interface Observer.
- f Les observateurs sont faiblement couplés dans le sens où le sujet ne sait rien d'eux, à part qu'ils implémentent l'interface Observateur.
- f Vous pouvez pousser ou extraire des données du sujet lorsque vous utilisez le modèle (le pousser est considéré comme plus « correct »).
- f Swing utilise largement le modèle Observer, comme le font de nombreux frameworks d'interface graphique.
- f Vous trouverez également le modèle dans de nombreux autres endroits, notamment RxJava, JavaBeans et RMI, ainsi que dans d'autres frameworks de langage, comme Cocoa, Swift et les événements JavaScript.
- f Le modèle d'observateur est lié au modèle de publication/abonnement, qui est destiné à des applications plus complexes.
- f situations avec plusieurs sujets et/ou plusieurs types de messages.
- f Le modèle Observer est un modèle couramment utilisé, et nous le reverrons lorsque nous découvrirons le modèle-vue-contrôleur.



Défi du principe de conception

Pour chaque principe de conception, décrivez comment le modèle Observer utilise le principe.

Principe de conception

Identifiez les aspects de votre application qui varient et séparez-les de ce qui reste le même.

Principe de conception

Programme vers une interface, pas une implémentation.

C'est une question difficile. Indice : pensez à la façon dont les observateurs

et les sujets travaillent ensemble.

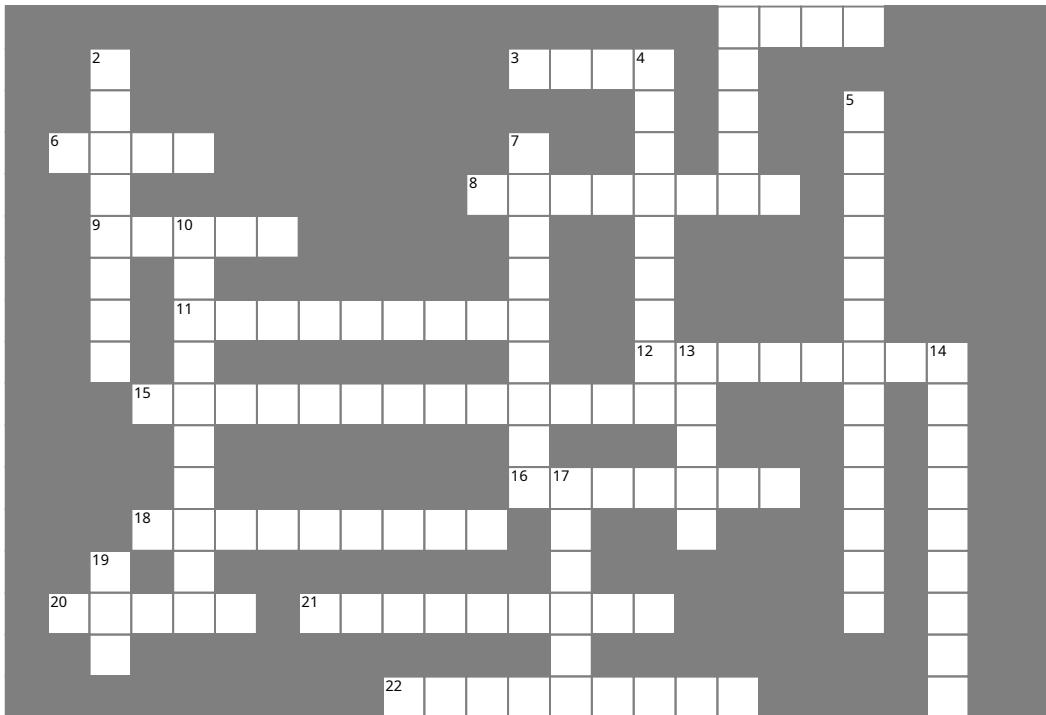
Principe de conception

Privilégiez la composition plutôt que l'héritage.



Mots croisés sur les modèles de conception

Il est temps de redonner du travail à votre cerveau droit !
Tous les mots-solutions sont issus des chapitres 1 et 2.



À TRAVERS

1. Un sujet aime parler à _____ observateurs.
3. Le sujet souhaitait initialement transmettre toutes les données à l'observateur.
6. Le PDG a presque oublié l'affichage de l'index _____.
8. CurrentConditionsDisplay implémente cette interface.
9. Framework Java avec de nombreux observateurs.
11. Un sujet est semblable à un _____.
12. Les observateurs aiment être _____ quand quelque chose de nouveau se produit.
15. Comment vous retirer de la liste des observateurs.
16. Lori était à la fois une observatrice et une _____.
18. Le sujet est un _____.
20. Vous souhaitez conserver votre accouplement _____.
21. Programme vers un _____ pas une implémentation.
22. Le Diable et l'Ange sont _____ au bouton.

VERS LE BAS

1. Il ne voulait plus d'ints, alors il s'est retiré.
2. Température, humidité et _____.
4. Le PDG de Weather-O-Rama porte le nom de ce type de tempête.
5. Il dit que tu devrais y aller.
7. Le sujet n'a pas besoin d'en savoir beaucoup sur le _____.
10. La classe WeatherData _____ l'interface Subject.
13. Ne comptez pas là-dessus pour recevoir une notification.
14. Les observateurs sont _____ sur le sujet.
17. Implémentez cette méthode pour être averti.
19. Jill en a eu un à elle.



Sharpen your pencil Solution

Sur la base de notre première mise en œuvre, lesquels des énoncés suivants s'appliquent ? (Choisissez toutes les réponses applicables.)

- ✓ A. Nous codons des implémentations concrètes, pas des interfaces.
- ✓ B. Pour chaque nouvel élément d'affichage, nous devons modifier le code.
- ✓ C. Nous n'avons aucun moyen d'ajouter des éléments d'affichage au moment de l'exécution.
- D. Les éléments d'affichage n'implémentent pas d'interface commune.
- ✓ E. Nous n'avons pas encore résumé ce qui change.
- F. Nous violons l'encapsulation de la classe WeatherData.



Conception Principe Défi Solution

Principe de conception

Identifiez les aspects de votre application qui varient et séparez-les de ce qui reste le même.

Ce qui varie dans le modèle d'observateur est l'état du sujet et le nombre et types d'observateurs. Avec ce modèle, vous pouvez faire varier les objets qui dépendent de l'état du sujet, sans avoir à changer cela

Sujet. C'est ce qu'on appelle planifier à l'avance !

Principe de conception

Programme vers une interface, pas une implémentation.

Le sujet et les observateurs utilisent tous deux des interfaces.

Le sujet garde la trace des objets implémentant

l'interface de l'observateur, tandis que les observateurs

inscrivez-vous et soyez averti par le sujet

interface. Comme nous l'avons vu, cela permet de garder les choses agréables et

couplé de manière lâche.

Principe de conception

Privilégiez la composition plutôt que l'héritage.

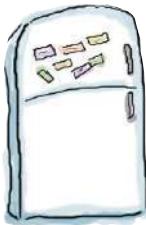
Le modèle Observer utilise la composition pour composer

n'importe quel nombre d'observateurs avec leur sujet.

Ces relations ne sont pas établies par une sorte de

de la hiérarchie d'héritage. Non, ils sont mis en place à

exécution par composition !



Solution Code Magnets

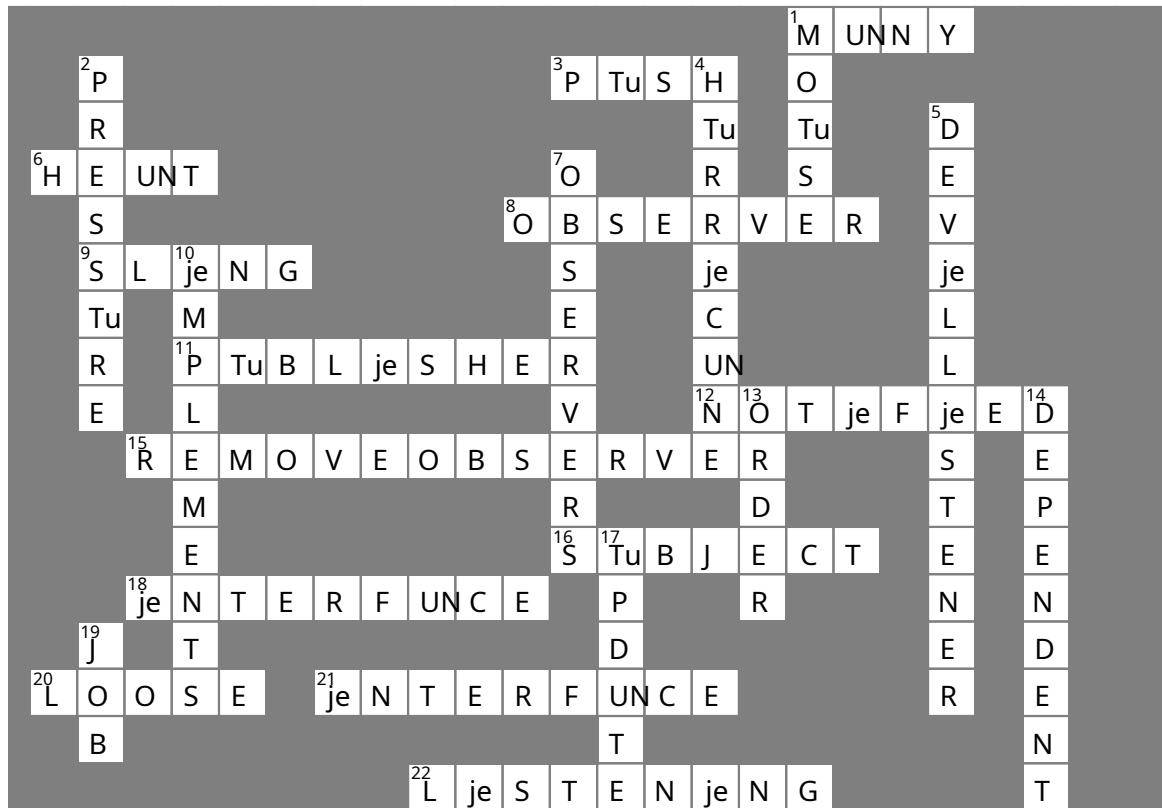
La classe ForecastDisplay est toute mélangée sur le réfrigérateur. Pouvez-vous reconstruire les extraits de code pour le faire fonctionner ? Certaines des accolades sont tombées par terre et elles étaient trop petites pour être ramassées, alors n'hésitez pas à en ajouter autant que vous le souhaitez ! Voici notre solution.

```
classe publique ForecastDisplay implémente  
Observer, DisplayElement {  
  
    flotteur privé currentPressure = 29,92f; flotteur  
    privé lastPressure;  
  
    Données météorologiques privées données météorologiques;  
  
    public ForecastDisplay(Données Météo  
    données météo) {  
  
        ceci.weatherData = donnéesmétéo;  
  
        weatherData.registerObserver(ceci);  
    }  
  
    public void mise à jour() {  
  
        dernière pression = pression actuelle; pression actuelle  
        = weatherData.getPressure();  
  
        afficher();  
    }  
  
    affichage public void() {  
        // afficher le code ici  
    }  
}
```



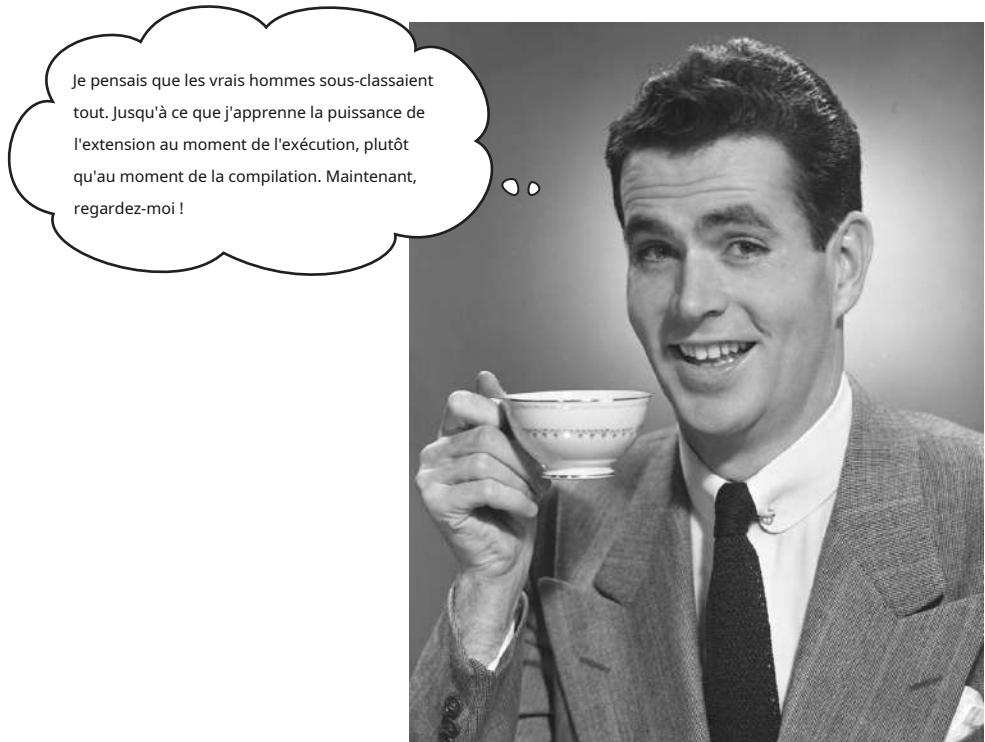
Conception

Solution de mots croisés



3le modèle de décorateur

* *Objets de décoration* *



Je pensais que les vrais hommes sous-classaient tout. Jusqu'à ce que j'apprenne la puissance de l'extension au moment de l'exécution, plutôt qu'au moment de la compilation. Maintenant, regardez-moi !

Appelez simplement ce chapitre « Design Eye for the Inheritance Guy ».

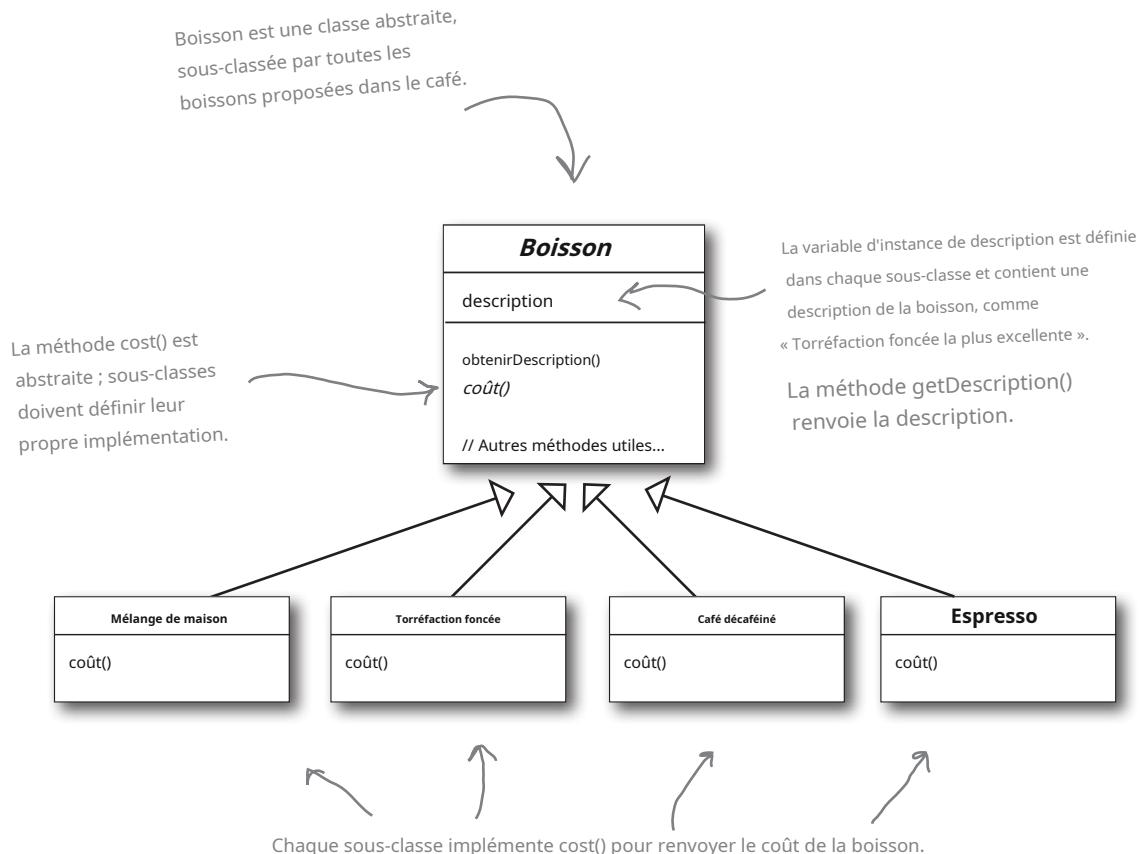
Nous allons réexaminer l'utilisation excessive typique de l'héritage et vous apprendrez à décorer vos classes au moment de l'exécution en utilisant une forme de composition d'objets. Pourquoi ? Une fois que vous connaissez les techniques de décoration, vous serez en mesure de donner à vos objets (ou à ceux de quelqu'un d'autre) de nouvelles responsabilités *sans apporter aucune modification de code aux classes sous-jacentes*.

Bienvenue chez Starbuzz Coffee

Starbuzz Coffee s'est fait un nom en tant que café à la croissance la plus rapide. Si vous en avez vu un au coin de votre rue, regardez de l'autre côté de la rue ; vous en verrez un autre.

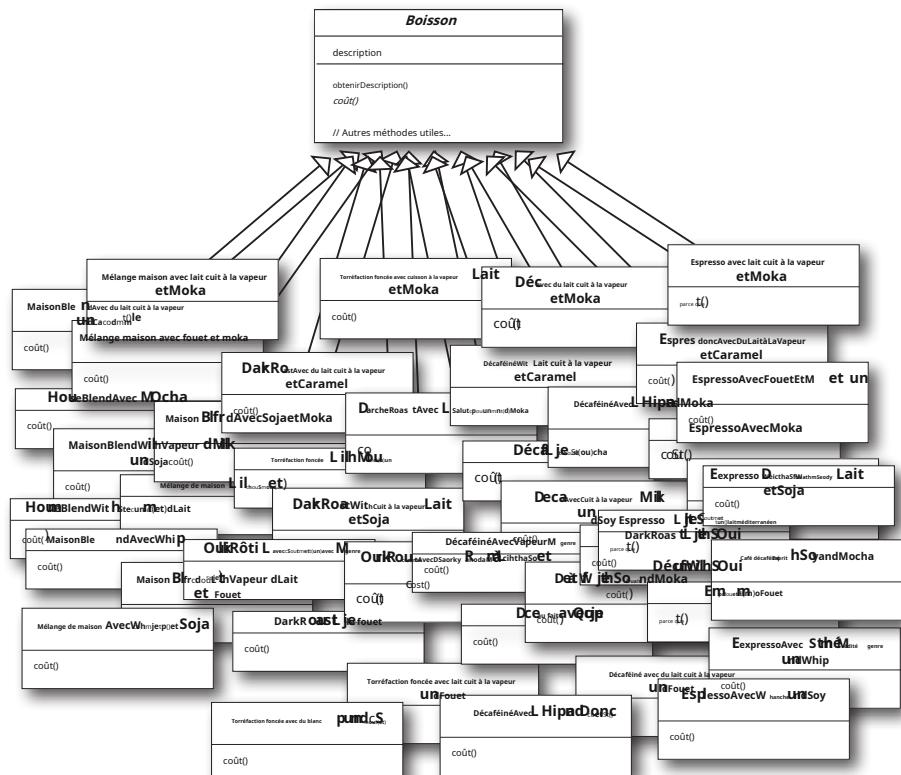
Parce qu'ils ont connu une croissance si rapide, ils s'efforcent de mettre à jour leurs systèmes de commande pour correspondre à leurs offres de boissons.

Lorsqu'ils ont démarré leur activité, ils ont conçu leurs cours comme ceci...



En plus de votre café, vous pouvez également demander plusieurs condiments comme du lait cuit à la vapeur, du soja et du moka (autrement appelé chocolat), et le tout accompagné de lait fouetté. Starbuzz facture un peu pour chaque condiment, il faut donc vraiment qu'ils les intègrent à leur système de commande.

Voici leur première tentative...



Ouah !
Peux-tu dire
« explosion de classe » ?

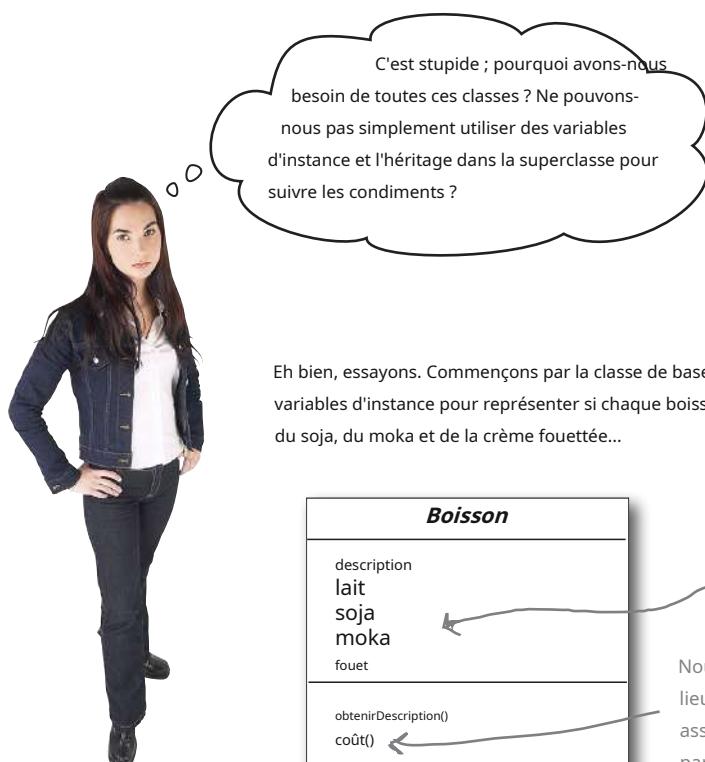
Chaque méthode de coût calcule le coût du café ainsi que des autres condiments de la commande.

violer principes de conception

Il est assez évident que Starbuzz s'est créé un cauchemar en matière de maintenance. Que se passe-t-il lorsque le prix du lait augmente ? Que fait-on lorsqu'on ajoute un nouveau nappage au caramel ?

En pensant au-delà du problème de maintenance, lesquels des principes de conception que nous avons abordés jusqu'à présent violent-ils ?

Indice : ils en violent deux de manière flagrante !



Eh bien, essayons. Commençons par la classe de base Beverage et ajoutons des variables d'instance pour représenter si chaque boisson contient ou non du lait, du soja, du moka et de la crème fouettée...

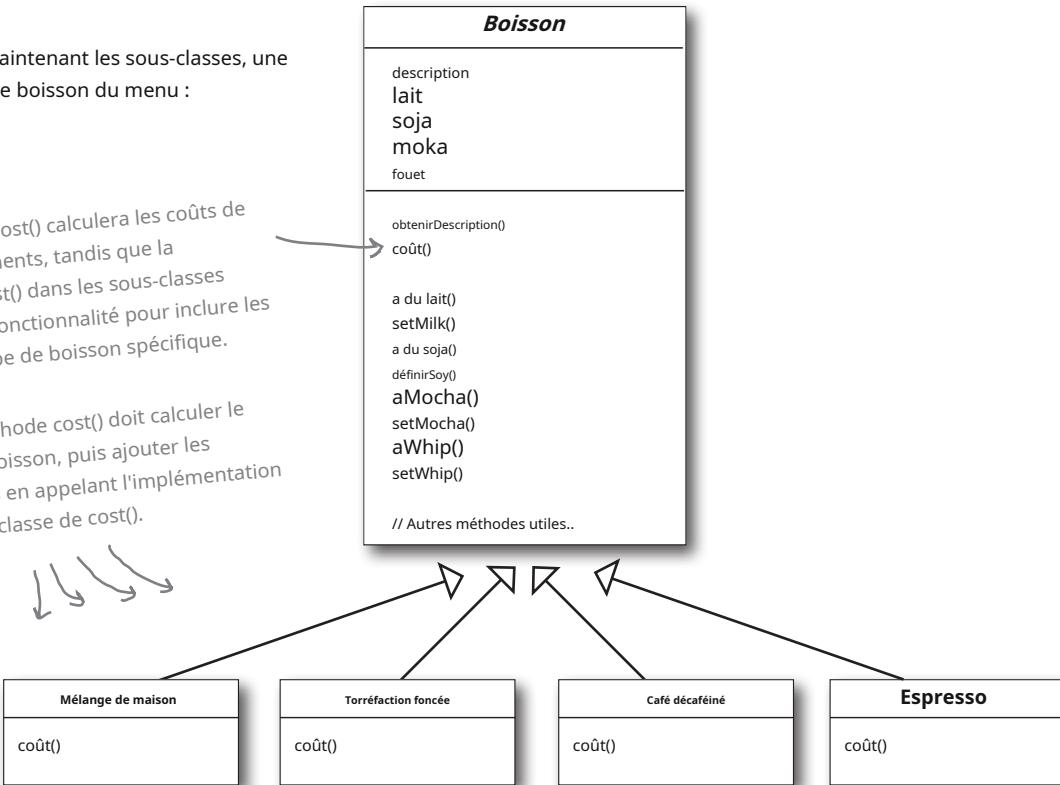
Boisson	
description	
lait	← Nouvelles valeurs booléennes pour chaque condiment.
soja	
moka	
fouet	
obtenirDescription()	
coût()	← Nous allons maintenant implémenter cost() dans Beverage (au lieu de le garder abstrait), afin qu'il puisse calculer les coûts associés aux condiments pour une instance de boisson particulière. Les sous-classes remplaceront toujours cost(), mais elles invoqueront également la version super afin de pouvoir calculer le coût total de la boisson de base plus les coûts des condiments ajoutés.
a du lait()	
setMilk()	
a du soja()	
définirSoy()	
aMocha()	
setMocha()	
aWhip()	
setWhip()	
// Autres méthodes utiles..	

Il obtiennent et définissent les valeurs booléennes pour les condiments.

Ajoutons maintenant les sous-classes, une pour chaque boisson du menu :

La superclasse cost() calculera les coûts de tous les condiments, tandis que la superclasse cost() dans les sous-classes étendra cette fonctionnalité pour inclure les coûts de ce type de boisson spécifique.

Chaque méthode cost() doit calculer le coût de la boisson, puis ajouter les condiments en appelant l'implémentation de la superclasse de cost().



Sharpen your pencil

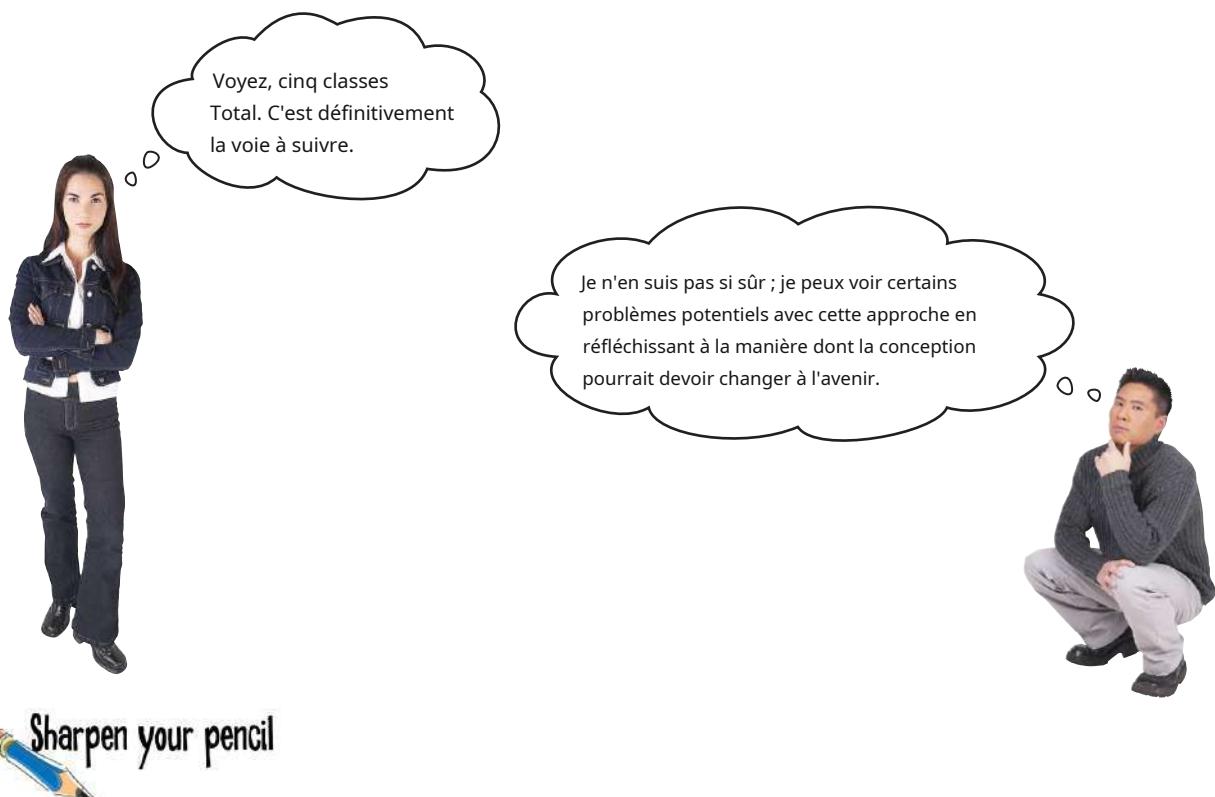
Écrivez les méthodes cost() pour les classes suivantes (le pseudo-Java est acceptable) :

```

classe publique Boisson {
    coût double public() {
        }
    }
  
```

```

classe publique DarkRoast étend Beverage {
    public DarkRoast() {
        description = "Torréfaction foncée la plus excellente";
    }
    coût double public() {
        }
    }
  
```



Sharpen your pencil

Quelles exigences ou autres facteurs pourraient changer et avoir un impact sur cette conception ?

Les changements de prix des condiments nous obligeront à modifier le code existant.

De nouveaux condiments nous obligeront à ajouter de nouvelles méthodes et à modifier la méthode de coût dans la superclasse.

Nous pourrions avoir de nouvelles boissons. Pour certaines de ces boissons (thé glacé ?), les condiments peuvent ne pas être appropriés, mais la sous-classe Tea héritera toujours de méthodes comme hasWhip().

Comme nous l'avons vu dans
Chapitre 1, c'est une
très mauvaise idée !

Et si un client veut un double moka ?

À ton tour :



Gourou et étudiant...

Gourou: Cela fait déjà un certain temps depuis notre dernière rencontre. Avez-vous médité sur l'héritage ?

Étudiant: Oui, Guru. Bien que l'héritage soit puissant, j'ai appris qu'il ne conduit pas toujours à la vie la plus flexible ou la plus heureuse. conceptions maintenables.

Gourou: Ah oui, tu as fait quelques progrès. Alors dis-moi, mon élève, comment vas-tu parvenir à la réutilisation si ce n'est par l'héritage ?

Étudiant: Gourou, j'ai appris qu'il existe des moyens d'« hériter » du comportement au moment de l'exécution grâce à la composition et à la délégation.

Gourou: S'il vous plaît, continuez...

Étudiant: Lorsque j'hérite d'un comportement par sous-classement, ce comportement est défini de manière statique au moment de la compilation. De plus, toutes les sous-classes doivent hériter du même comportement. Si, toutefois, je peux étendre le comportement d'un objet par composition, je peux le faire de manière dynamique au moment de l'exécution.

Gourou: Très bien, vous commencez à voir le pouvoir de la composition.

Étudiant: Oui, il m'est possible d'ajouter plusieurs nouvelles responsabilités aux objets grâce à cette technique, y compris des responsabilités auxquelles le concepteur de la superclasse n'avait même pas pensé. Et je n'ai pas besoin de toucher à leur code !

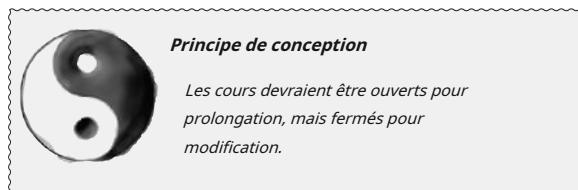
Gourou: Qu'avez-vous appris sur l'effet de la composition sur la maintenance de votre code ?

Étudiant: Eh bien, c'est ce que je voulais dire. En composant dynamiquement des objets, je peux ajouter de nouvelles fonctionnalités en écrivant du nouveau code plutôt qu'en modifiant le code existant. Comme je ne modifie pas le code existant, les risques d'introduire des bugs ou de provoquer des effets secondaires imprévus dans le code préexistant sont considérablement réduits.

Gourou: Très bien. Assez pour aujourd'hui. J'aimerais que vous alliez méditer davantage sur ce sujet... Souvenez-vous, le code doit être fermé (au changement) comme la fleur de lotus le soir, mais ouvert (à l'extension) comme la fleur de lotus le matin.

Le principe ouvert-fermé

Nous abordons l'un des principes de conception les plus importants :



Entrez, nous sommes ouvrir. Frais libres d'étendre notre classes avec n'importe quel nouveau comportement que vous souhaitez. Si vos besoins ou vos exigences changent (et nous savons qu'ils changeront), allez-y et créez vos propres extensions.



Désolé, nous sommes fermé. C'est vrai, nous j'ai passé beaucoup de temps à obtenir cela Le code est correct et sans bug, nous ne pouvons donc pas vous laisser modifier le code existant. Il doit rester fermé à la modification. Si vous ne l'aimez pas, vous pouvez en parler au responsable.

Notre objectif est de permettre aux classes d'être facilement étendues pour intégrer de nouveaux comportements sans modifier le code existant. Qu'obtiendrons-nous si nous y parvenons ? Des conceptions résilientes au changement et suffisamment flexibles pour intégrer de nouvelles fonctionnalités afin de répondre aux exigences changeantes.

there are no Dumb Questions

Q:

Ouvert à l'extension et fermé à la modification ? Cela semble très contradictoire.

Comment une conception peut-elle être les deux ?

UN:

C'est une très bonne question. Cela semble contradictoire au premier abord. Après tout, moins quelque chose est modifiable, plus il est difficile de l'étendre, n'est-ce pas ?

Il s'avère cependant qu'il existe des techniques OO astucieuses permettant d'étendre les systèmes, même si nous ne pouvons pas modifier le code sous-jacent. Pensez au modèle Observer (au chapitre 2)... en ajoutant de nouveaux observateurs, nous pouvons étendre le sujet à tout moment, sans ajouter de code au sujet. Vous verrez de nombreuses autres façons d'étendre le comportement avec d'autres techniques de conception OO.

Q:

D'accord, je comprends Observer, mais comment puis-je généralement concevoir quelque chose qui soit extensible mais fermé à la modification ?

UN:

De nombreux modèles nous offrent des conceptions éprouvées qui protègent votre code contre toute modification en fournissant un moyen d'extension. Dans ce chapitre, vous verrez un bon exemple d'utilisation du modèle Decorator pour suivre le principe ouvert-fermé.

Q:

Comment puis-je faire en sorte que chaque partie

de ma conception suive le principe ouvert-fermé ?

UN:

En général, ce n'est pas possible. Rendre la conception OO flexible et ouverte aux extensions sans modifier le code existant demande du temps et des efforts. En général, nous ne pouvons pas nous permettre de limiter chaque partie de nos conceptions (et ce serait probablement du gaspillage). Le respect du principe ouvert-fermé introduit généralement de nouveaux niveaux d'abstraction, ce qui ajoute de la complexité à notre code. Vous devez vous concentrer sur les domaines les plus susceptibles de changer dans vos conceptions et y appliquer les principes.

Q:

Comment puis-je savoir quels domaines de changement sont les plus importants ?

UN:

C'est en partie une question d'expérience dans la conception de systèmes OO et également une question de connaissance du domaine dans lequel vous travaillez. Regarder d'autres exemples vous aidera à apprendre à identifier les domaines de changement dans vos propres conceptions.

Même si cela peut sembler contradictoire, il existe des techniques permettant d'étendre le code sans modification directe.

Soyez prudent lorsque vous choisissez les zones de code qui doivent être étendues ; appliquer le principe ouvert-fermé PARTOUT est un gaspillage et inutile, et peut conduire à un code complexe et difficile à comprendre.

Découvrez le modèle Decorator

D'accord, nous avons vu que représenter nos boissons et condiments avec l'héritage n'a pas très bien fonctionné : nous obtenons des explosions de classes et des conceptions rigides, ou nous ajoutons des fonctionnalités à la classe de base qui ne sont pas appropriées pour certaines des sous-classes.

Alors, voici ce que nous allons faire à la place : nous commencerons par une boisson et la « décorerons » avec les condiments au moment de l'exécution. Par exemple, si le client veut un café torréfié foncé avec du moka et de la crème fouettée, nous allons :

- 1 Commencez avec un objet DarkRoast.**
- 2 Décorez-le avec un objet Moka.**
- 3 Décorez-le avec un objet Fouet.**
- 4 Appelez la méthode cost() et comptez sur la délégation pour additionner les coûts des condiments.**

D'accord, mais comment « décorer » un objet et comment la délégation intervient-elle dans tout cela ?

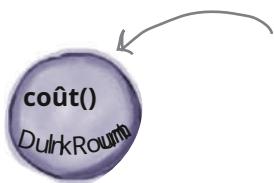
Un conseil : considérez les objets décorateurs comme des « enveloppes ». Voyons comment cela fonctionne...

Bon, assez parlé du « Club de conception orientée objet ». Nous avons de vrais problèmes ici ! Vous vous souvenez de nous ? Starbuzz Coffee ? Pensez-vous que vous pourriez utiliser certains de ces principes de conception pour réellement aidez nous?



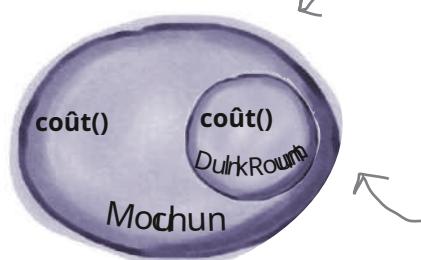
Créer une commande de boissons avec Decorators

- 1** Nous commençons avec notre objet DarkRoast.



N'oubliez pas que DarkRoast hérite de Beverage et possède une méthode cost() qui calcule le coût de la boisson.

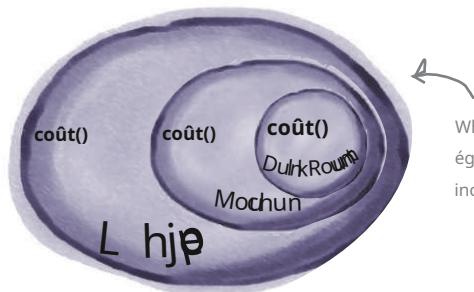
- 2** Le client veut du Moka, nous créons donc un objet Moka et l'enroulons autour du DarkRoast.



L'objet Mocha est un décorateur. Son type reflète l'objet qu'il décore, dans ce cas, une boisson. (Par « miroir », nous entendons qu'il s'agit du même type.)

Ainsi, Mocha possède également une méthode cost() et, grâce au polymorphisme, nous pouvons également traiter n'importe quelle boisson enveloppée dans Mocha comme une boisson (car Mocha est un sous-type de boisson).

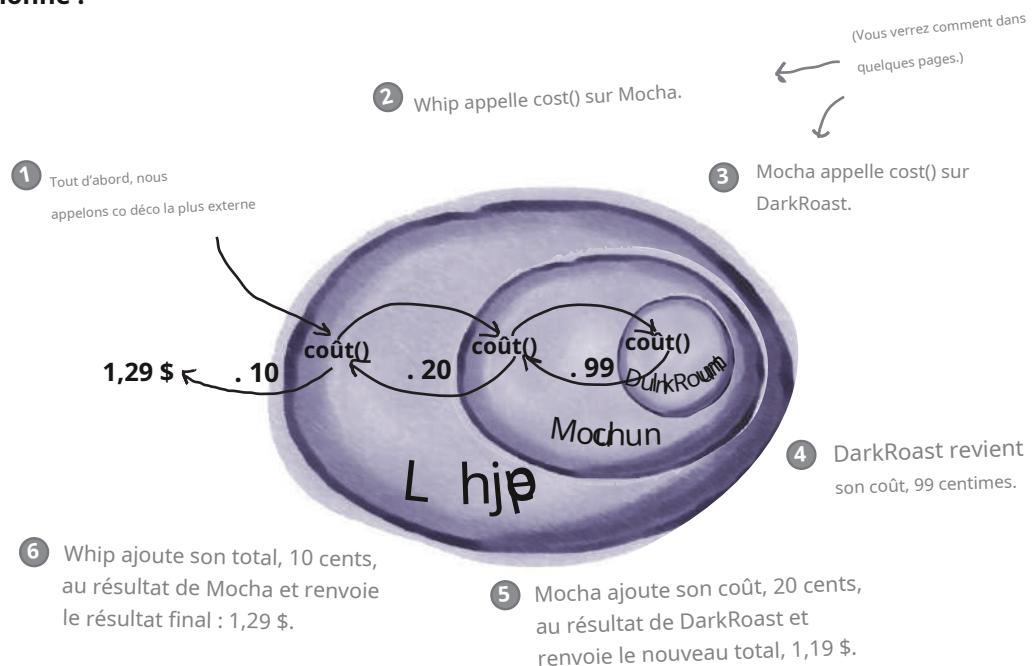
- 3** Le client veut également du Whip, nous créons donc un décorateur Whip et emballons du Moka avec.



Whip est un décorateur, il reflète donc également le type de DarkRoast et inclut une méthode cost().

Ainsi, un DarkRoast enveloppé de Moka et de Whip est toujours une boisson et nous pouvons en faire tout ce que nous pouvons faire avec un DarkRoast, y compris appeler sa méthode cost().

- 4 Il est maintenant temps de calculer le coût pour le client. Pour cela, nous appelons `cost()` sur le décorateur le plus externe, `Whip`, et `Whip` va déléguer le calcul du coût aux objets qu'il décore. Et ainsi de suite. Voyons comment cela fonctionne :



Bon, voici ce que nous savons sur les décorateurs, jusqu'à présent...

- f Les décorateurs ont le même supertype que les objets qu'ils décorent.
 - f Vous pouvez utiliser un ou plusieurs décorateurs pour envelopper un objet.
 - f Étant donné que le décorateur a le même supertype que l'objet qu'il décore, nous pouvons transmettre un objet décoré à la place de l'objet d'origine (emballé).
 - f Le décorateur ajoute son propre comportement avant et/ou après avoir délégué à l'objet qu'il décore le soin de faire le reste du travail.
 - f Les objets peuvent être décorés à tout moment, nous pouvons donc décorer des objets de manière dynamique au moment de l'exécution avec autant de décorateurs que nous le souhaitons.
- Point clé !

Voyons maintenant comment tout cela fonctionne réellement en regardant la définition du modèle Decorator et en écrivant du code.

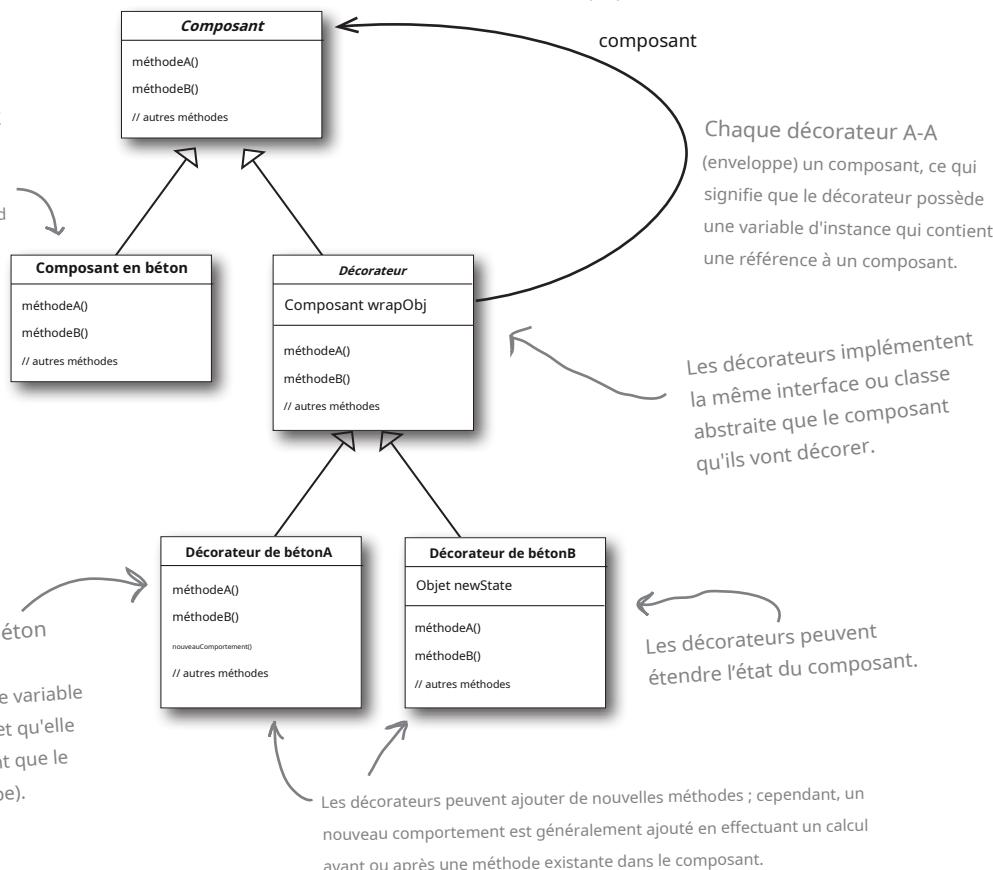
Le modèle de décorateur défini

Commençons par examiner la description du modèle Decorator :

Le modèle de décorateur attache des responsabilités supplémentaires à un objet de manière dynamique. Les décorateurs offrent une alternative flexible à la sous-classification pour étendre les fonctionnalités.

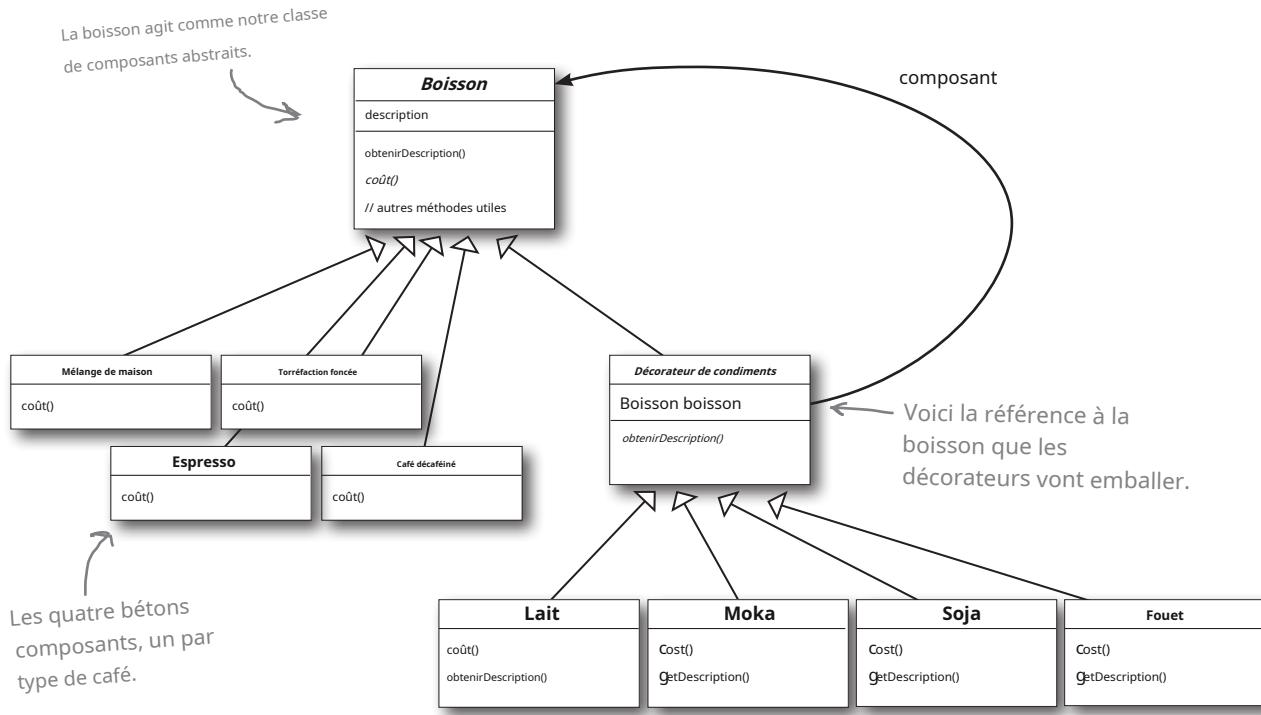
Bien que cela décrive le rôle du modèle Decorator, cela ne nous donne pas beaucoup d'informations sur la façon dont nous le ferions appliquer le modèle de notre propre implémentation. Jetons un œil au diagramme de classes, qui est un peu plus révélateur (sur la page suivante, nous examinerons la même structure appliquée au problème des boissons).

Le composant concret est l'objet auquel nous allons ajouter dynamiquement un nouveau comportement. Il étend Component.



Décorer nos boissons

Retravaillons nos boissons Starbuzz en utilisant le modèle Decorator...



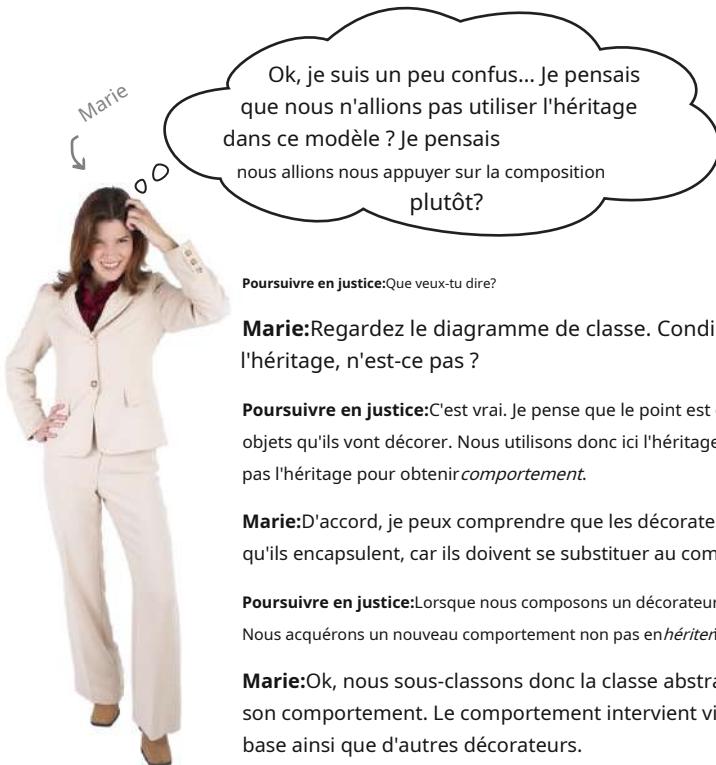
Et voici nos décorateurs de condiments ; remarquez qu'ils doivent implémenter non seulement cost() mais aussi getDescription(). Nous verrons pourquoi dans un instant...



Avant d'aller plus loin, réfléchissez à la manière dont vous implémenteriez la méthode **cost()** des cafés et des condiments. Réfléchissez également à la manière dont vous implémenteriez la méthode **getDescription()** des condiments.

Conversation en cabine

Une certaine confusion entre héritage et composition



Poursuivre en justice: Que veux-tu dire?

Marie: Regardez le diagramme de classe. CondimentDecorator étend la classe Beverage. C'est de l'héritage, n'est-ce pas ?

Poursuivre en justice: C'est vrai. Je pense que le point est qu'il est essentiel que les décorateurs aient le même type que les objets qu'ils vont décorer. Nous utilisons donc ici l'héritage pour atteindre la *correspondance de type*, mais nous n'utilisons pas l'héritage pour obtenir *comportement*.

Marie: D'accord, je peux comprendre que les décorateurs aient besoin de la même « interface » que les composants qu'ils encapsulent, car ils doivent se substituer au composant. Mais où intervient le comportement ?

Poursuivre en justice: Lorsque nous composons un décorateur avec un composant, nous ajoutons un nouveau comportement. Nous acquérons un nouveau comportement non pas en héritier il s'agit d'une superclasse, mais par composition objets ensemble.

Marie: Ok, nous sous-classons donc la classe abstraite Beverage afin d'avoir le bon type, et non pour hériter de son comportement. Le comportement intervient via la composition des décorateurs avec les composants de base ainsi que d'autres décorateurs.

Poursuivre en justice: C'est exact.

Marie: Oh, je comprends ! Et comme nous utilisons la composition d'objets, nous avons beaucoup plus de flexibilité quant à la façon de mélanger et d'associer les condiments et les boissons. Très astucieux.

Poursuivre en justice: Oui, si nous nous appuyons sur l'héritage, notre comportement ne peut être déterminé que de manière statique au moment de la compilation. En d'autres termes, nous obtenons uniquement le comportement que la superclasse nous donne ou que nous remplaçons. Avec la composition, nous pouvons mélanger et assortir les décorateurs comme nous le souhaitons... *au moment de l'exécution*.

Marie: Je comprends : nous pouvons implémenter de nouveaux décorateurs à tout moment pour ajouter un nouveau comportement. Si nous nous appuyons sur l'héritage, nous devrions intervenir et modifier le code existant à chaque fois que nous voudrions un nouveau comportement.

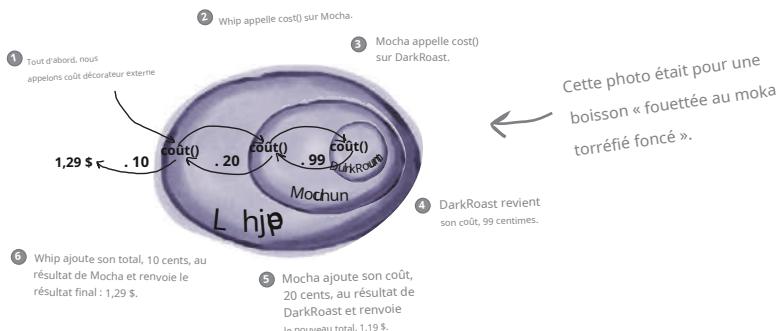
Poursuivre en justice: Exactement.

Marie: J'ai juste une autre question : si tout ce dont nous avons besoin pour hériter est le type du composant, comment se fait-il que nous n'ayons pas utilisé une interface au lieu d'une classe abstraite pour la classe Beverage ?

Poursuivre en justice: Eh bien, rappelez-vous, lorsque nous avons reçu ce code, Starbuzz l'avait déjà *avait* une classe de boisson abstraite. Traditionnellement, le modèle Decorator spécifie un composant abstrait, mais en Java, nous pourrions évidemment utiliser une interface. Mais nous essayons toujours d'éviter de modifier le code existant, alors ne le « réparez » pas si la classe abstraite fonctionne parfaitement.

Nouvelle formation de barista

Faites une image de ce qui se passe lorsque la commande concerne une boisson « double moka soy latte with whip ». Utilisez le menu pour obtenez les prix corrects et dessinez votre image en utilisant le même format que nous avons utilisé plus tôt (à partir de quelques pages en arrière) :



Sharpen your pencil

Dessine ton image ici.

Ok, j'ai besoin que tu me fasses un double moka latte au soja avec de la crème fouettée.



Ecrire le code Starbuzz

Il est temps de transformer cette conception en code réel.

Commençons par la classe Beverage, qui n'a pas besoin d'être modifiée par rapport à la conception originale de Starbuzz. Jetons un œil :



```
classe abstraite publique Boisson {
    Description de la chaîne = "Boisson inconnue";
    chaîne publique getDescription() {
        description du retour;
    }
    public abstrait double coût();
}
```

Beverage est une classe abstraite avec les deux méthodes getDescription() et cost().

getDescription est déjà implémenté pour nous, mais nous devons implémenter cost() dans les sous-classes.

La boisson est assez simple. Implémentons également la classe abstraite pour les condiments (le décorateur) :

```
classe abstraite publique CondimentDecorator étend la gamme de boissons {
    Boisson boisson;
    chaîne publique abstrait getDescription();
```

Tout d'abord, nous devons être interchangeables avec une boisson, nous étendons donc la classe Boisson.

Voici la boisson que chaque décorateur emballera. Notez que nous utilisons le Supertype de boisson pour faire référence à la boisson afin que le décorateur puisse emballer n'importe quelle boisson.

Nous allons également exiger que le condiment les décorateurs réimplémentent tous la méthode getDescription(). Encore une fois, nous verrons pourquoi dans une seconde...

Codage des boissons

Maintenant que nous avons défini nos classes de base, implémentons quelques boissons. Nous commencerons par Espresso. N'oubliez pas que nous devons définir une description pour la boisson spécifique et également implémenter la méthode cost().

la classe publique Espresso étend Beverage {

```

public Espresso() {
    description = "Expresso";
}

coût double public() {
    retour 1,99 ;
}

```

Nous étendons d'abord la classe Boisson, puisqu'il s'agit d'une boisson.

Pour prendre en charge la description, nous la définissons dans le constructeur de la classe. N'oubliez pas que la variable d'instance de description est héritée de Beverage.

Enfin, nous devons calculer le coût d'un expresso. Nous n'avons pas besoin de nous soucier d'ajouter des condiments dans cette classe, nous devons simplement renvoyer le prix d'un expresso : 1,99 \$.

classe publique HouseBlend étend Beverage {

```

public HouseBlend() {
    description = "Café mélange maison";
}

coût double public() {
    retour .89;
}

```

Bon, voici une autre boisson. Il suffit de définir la description appropriée, « Café maison », puis de renvoyer le prix correct : 89 ¢.

Café Starbuzz	
Cafés	.89
Mélange maison	.99
Torréfaction foncée	1,05
Café décaféiné	1,99
Espresso	.89
Condiments	
Lait cuit à la vapeur	.20
Moka	.15
Soja	.10
Fouet	

Vous pouvez créer les deux autres classes de boissons (DarkRoast et Decaf) exactement de la même manière.

Codage des condiments

Si vous regardez le diagramme de classe du modèle Decorator, vous verrez que nous avons maintenant écrit notre composant abstrait (Beverage), que nous avons nos composants concrets (HouseBlend) et que nous avons notre décorateur abstrait (CondimentDecorator). Il est maintenant temps d'implémenter les décorateurs concrets. Voici Mocha :

```

Mocha est un décorateur, nous étendons donc CondimentDecorator.
classe publique Mocha étend CondimentDecorator {
    Moka public (boisson) {
        cette.boisson = boisson;
    }
    chaîne publique getDescription() {
        retourner boisson.getDescription() + ", Moka";
    }
    coût double public() {
        retourner boisson.coût() + .20;
    }
}

Il nous faut maintenant calculer le coût de notre boisson avec Mocha. Tout d'abord, nous déléguons l'appel à l'objet que nous décorons afin qu'il puisse calculer le coût ; ensuite, nous ajoutons le coût de Mocha au résultat.

```

N'oubliez pas que CondimentDecorator étend Beverage.

Nous allons instancier Mocha avec une référence à une boisson.

N'oubliez pas que cette classe hérite de la variable d'instance Beverage pour contenir la boisson que nous emballons.

Nous définissons cette variable d'instance sur l'objet que nous enveloppons. Ici, nous transmettons la boisson que nous enveloppons au constructeur du décorateur.

Nous voulons que notre description inclue non seulement la boisson (par exemple « Torréfaction foncée »), mais aussi chaque élément qui la décore (par exemple « Torréfaction foncée, Moka »). Nous déléguons donc d'abord à l'objet que nous décorons la tâche d'obtenir sa description, puis nous ajoutons « Moka » à cette description.

Sur la page suivante, nous allons réellement instancier la boisson et l'envelopper avec tous ses condiments (décorateurs), mais d'abord...



Écrivez et compilez le code des autres condiments Soy and Whip. Vous en aurez besoin pour terminer et tester l'application.

Servir quelques cafés

Félicitations. Il est temps de vous asseoir, de commander quelques cafés et de vous émerveiller devant le design flexible que vous avez créé avec le modèle Decorator.

Voici un code de test pour passer des commandes :

```
classe publique StarbuzzCoffee {  
  
    public static void main(String args[]) {  
        Boisson boisson = new Espresso();  
        System.out.println(beverage.getDescription()  
            + " $" + beverage.cost());  
  
        Boisson boisson2 = new DarkRoast();  
        boisson2 = nouveau Moka(boisson2);  
        boisson2 = nouveau Moka(boisson2);  
        boisson2 = nouveau Whip(boisson2);  
        System.out.println(beverage2.getDescription()  
            + " $" + beverage2.cost());  
  
        Boisson boisson3 = new HouseBlend();  
        boisson3 = nouveau Soy(boisson3);  
        boisson3 = nouveau Moka(boisson3);  
        boisson3 = nouveau Whip(boisson3);  
        System.out.println(beverage3.getDescription()  
            + " $" + beverage3.cost());  
    }  
}
```

Commandez un espresso, sans condiments, et imprimez sa description et son prix.

Créez un objet DarkRoast.

Enveloppez-le d'un moka.

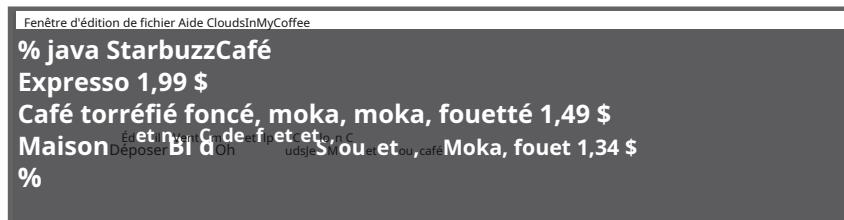
Enveloppez-le dans un deuxième Moka.

Enveloppez-le dans un Whip.

Enfin, donnez-nous un HouseBlend avec du soja, du moka et de la crème fouettée.

Maintenant, passons ces commandes :

* Nous allons voir une bien meilleure façon de créer des objets décorés lorsque nous aborderons les modèles de conception Factory et Builder. Veuillez noter que le modèle Builder est abordé dans l'annexe.



```
Fenêtre d'édition de fichier Aide CloudsInMyCoffee  
% java StarbuzzCafé  
Expresso 1,99 $  
Café torréfié foncé, moka, moka, fouetté 1,49 $  
Maison Déposer BI 8 Oh Ed et un C de fip et eton C  
%ou et ou, café Moka, fouet 1,34 $  
%
```

there are no
Dumb Questions

Q:

Je suis un peu inquiet à propos du code qui pourrait tester un composant concret spécifique (par exemple, HouseBlend) et faire quelque chose, comme émettre une remise. Une fois que j'aurai enveloppé HouseBlend avec des décorateurs, cela ne fonctionnera plus.

UN:

C'est tout à fait exact. Si vous avez du code qui s'appuie sur le type du composant concret, les décorateurs briseront ce code. Tant que vous n'écrivez que du code par rapport au type de composant abstrait, l'utilisation de décorateurs restera transparente pour votre code. Cependant, une fois que vous aurez commencé à écrire du code par rapport aux composants concrets, vous devrez repenser la conception de votre application et votre utilisation des décorateurs.

Q:

Ne serait-il pas facile pour un client d'une boisson de se retrouver avec un décorateur qui n'est pas le décorateur le plus externe ? Par exemple, si j'avais un DarkRoast avec du Moka, du Soja et de la Crème Fouettée, il serait facile d'écrire du code qui finirait par faire référence à du Soja au lieu de Crème Fouettée, ce qui signifie qu'il n'inclurait pas de Crème Fouettée dans la commande.

UN:

Vous pourriez certainement argumenter que vous devez gérer plus d'objets avec le modèle Decorator et qu'il y a donc un risque accru que des erreurs de codage introduisent le type de problèmes que vous suggérez. Cependant, nous créons généralement des décorateurs en utilisant d'autres modèles comme Factory et Builder. Une fois que nous aurons couvert ces modèles, vous verrez que la création du composant concret avec son décorateur est « bien encapsulée » et ne conduit pas à ce genre de problèmes.

Q:

Les décorateurs peuvent-ils connaître les autres décos de la chaîne ? Supposons que je souhaite que ma méthode getDescription() imprime « Whip, Double Mocha » au lieu de « Mocha, Whip, Mocha ». Cela nécessiterait que mon décorateur le plus extérieur connaît tous les décorateurs qu'il enveloppe.

UN:

Les décorateurs sont censés ajouter un comportement à l'objet qu'ils enveloppent. Lorsque vous devez jeter un œil à plusieurs couches de la chaîne de décorateurs, vous commencez à pousser le décorateur au-delà de sa véritable intention. Néanmoins, de telles choses sont possibles. Imaginez un décorateur CondimentPrettyPrint qui analyse la description finale et peut imprimer « Moka, Whip, Moka » sous la forme « Whip, Double Mocha ». Notez que getDescription() pourrait renvoyer une ArrayList de descriptions pour faciliter cette opération.



Sharpen your pencil

Nos amis de Starbuzz ont introduit des tailles dans leur menu. Vous pouvez désormais commander un café dans les tailles tall, grande et venti (traduction : petit, moyen et grand). Starbuzz a vu cela comme une partie intrinsèque de la classe coffee, ils ont donc ajouté deux méthodes à la classe Beverage : setSize() et getSize(). Ils aimeraient également que les condiments soient facturés en fonction de la taille, ainsi, par exemple, le soja coûte respectivement 10¢, 15¢ et 20¢ pour les cafés tall, grande et venti. La classe Beverage mise à jour est présentée ci-dessous.

Comment modifieriez-vous les classes de décorateur pour gérer ce changement d'exigences ?

classe abstraite publique Boisson {

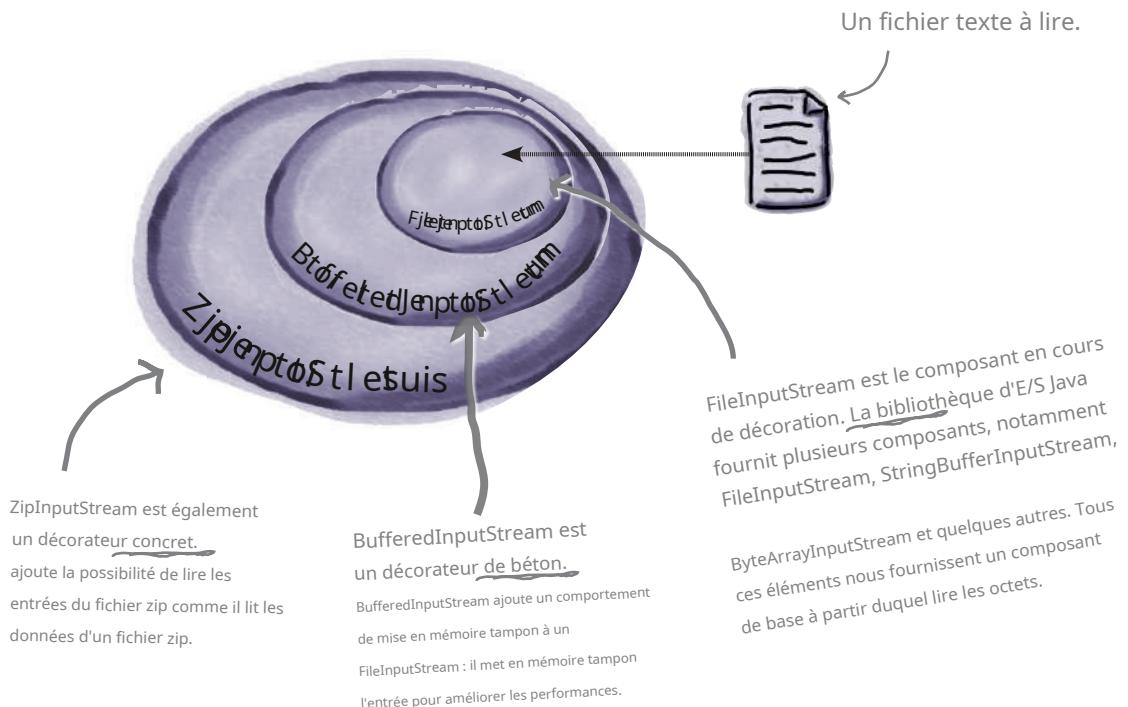
```

public enum Size { TALL, GRANDE, VENTI }; Taille
size = Size.TALL;
Description de la chaîne = "Boisson inconnue";
public String getDescription() {
    description du retour;
}
public void setSize(Taille taille) {
    ceci.taille = taille;
}
Taille publique getSize() {
    renvoie ceci.size;
}
public abstrait double coût();
}
```

Décorateurs du monde réel : E/S Java

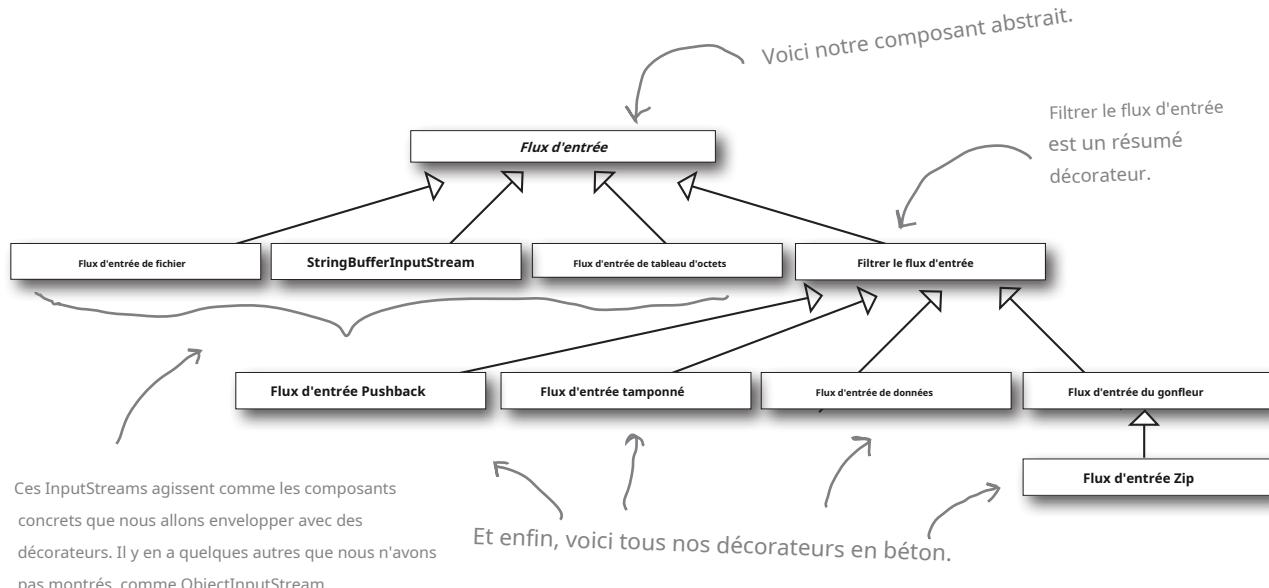
Le grand nombre de classes dans le package `java.io` est... accablant. Ne vous sentez pas seul si vous avez dit « ouah » la première (et la deuxième et la troisième) fois que vous avez regardé cette API.

Mais maintenant que vous connaissez le modèle Decorator, les classes d'E/S devraient avoir plus de sens puisque le package `java.io` est largement basé sur Decorator. Voici un ensemble typique d'objets qui utilisent des décorateurs pour ajouter des fonctionnalités à la lecture de données à partir d'un fichier :



Décorer les classes java.io

BufferedInputStream et ZipInputStream étendent tous deux FilterInputStream, qui étend InputStream. InputStream agit comme une classe de décorateur abstrait :



Vous pouvez voir que ce n'est pas si différent de la conception de Starbuzz. Vous devriez maintenant être en bonne position pour consulter les documents de l'API java.io et composer des décorateurs sur les différents *saïsirrusseaux*.

Vous verrez que les *sorties* flux ont la même conception. Et vous avez probablement déjà constaté que les flux Reader/Writer (pour les données basées sur des caractères) reflètent étroitement la conception des classes de flux (avec quelques différences et incohérences, mais suffisamment proches pour comprendre ce qui se passe).

Java I/O souligne également l'un des *inconvénients* du modèle Decorator : les conceptions utilisant ce modèle génèrent souvent un grand nombre de petites classes qui peuvent être écrasantes pour un développeur essayant d'utiliser l'API basée sur Decorator. Mais maintenant que vous savez comment fonctionne Decorator, vous pouvez garder les choses en perspective et lorsque vous utilisez l'API Decorator lourde de quelqu'un d'autre, vous pouvez comprendre comment leurs classes sont organisées afin de pouvoir facilement utiliser l'encapsulation pour obtenir le comportement que vous recherchez.

Écrire votre propre décorateur d'E/S Java

OK, vous connaissez le modèle de décorateur et vous avez vu le diagramme de classe d'E/S. Vous devriez être prêt à écrire votre propre décorateur d'entrée.

Que diriez-vous de ceci : écrivez un décorateur qui convertit tous les caractères majuscules en minuscules dans le flux d'entrée. En d'autres termes, si nous lisons « Je connais le modèle de décorateur donc je RÈGLE ! », alors votre décorateur convertit cela en « Je connais le modèle de décorateur donc je RÈGLE !

N'oubliez pas d'importer
java.io... (non affiché).

Tout d'abord, étendez le FilterInputStream, le décorateur abstrait pour tous les InputStreams.

classe publique LowerCaseInputStream étend FilterInputStream {

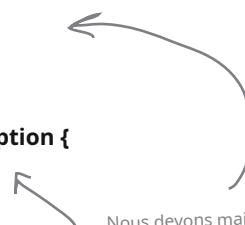
```
public LowerCaseInputStream(InputStream dans) {  
    super(dans);  
}
```

```
public int read() lance une exception IOException {  
    int c = dans.lire();  
    retour (c == -1 ? c : Caractère.toLowerCase((char)c));  
}
```

```
public int read(byte[] b, int offset, int len) lève une exception IOException {  
    int résultat = in.read(b, offset, len);  
    pour (int i = décalage; i < décalage+résultat; i++) {  
        b[i] = (octet)Caractère.en minuscules((char)b[i]);  
    }  
    retourner le résultat;  
}
```

N'OUBLIEZ PAS : nous ne fournissons pas d'instructions d'importation et de package dans les listes de codes. Obtenez le code source complet à partir de <https://wickedlysmart.com/head-first-design-patterns>.

Pas de problème. Je
il suffit d'étendre la classe
FilterInputStream et de
remplacer les méthodes read().



Nous devons maintenant implémenter deux méthodes de lecture. Elles prennent un octet (ou un tableau d'octets) et convertissent chaque octet (qui représente un caractère) en minuscule s'il s'agit d'un caractère majuscule.

Testez votre nouveau décorateur d'E/S Java

Écrivez un peu de code rapide pour tester le décorateur d'E/S :

```

classe publique InputTest {
    public static void main(String[] args) lève une exception IOException {
        int c;
        essayer {
            InputStream dans =
                nouveau LowerCaseInputStream(
                    nouveau BufferedInputStream(
                        nouveau FileInputStream("test.txt")));

            tandis que((c = dans.read()) >= 0) {
                Système.out.print((char)c);
            }

            joindre();
        } attraper (IOException e){
            e.printStackTrace();
        }
    }
}

```

Utilisez simplement le flux pour lire les caractères jusqu'à la fin du fichier et imprimez au fur et à mesure.

Configurez le FileInputStream et décitez-le, d'abord avec un BufferedInputStream, puis avec notre tout nouveau Filtre LowerCaseInputStream.

Je connais le modèle Decorator donc je RÈGNE !

fichier test.txt

Vous devez créer ce fichier.

Faites-le tourner :

```

Fenêtre d'édition de fichier Aide DécorateursRègle
% java InputTest
je connais le modèle de décoration donc je gouverne ! %

```



Les modèles dévoilés

Cette semaineentretien:
Confessions d'une décoratrice

Tête la première :Bienvenue, Decorator Pattern. Nous avons entendu dire que vous étiez un peu déprimé ces derniers temps ?

Décorateur:Oui, je sais que le monde me voit comme un modèle de design glamour, mais vous savez, j'ai ma part de problèmes, comme tout le monde.

Tête la première :Pouvez-vous peut-être partager avec nous certains de vos problèmes ?

Décorateur:Bien sûr. Eh bien, vous savez que j'ai le pouvoir d'ajouter de la flexibilité aux conceptions, c'est sûr, mais j'ai aussi un côté *obscur*. Vous voyez, je peux parfois ajouter beaucoup de petites classes à une conception, et cela donne parfois lieu à une conception qui est moins simple à comprendre pour les autres.

Tête la première :Pouvez-vous nous donner un exemple ?

Décorateur:Prenons les bibliothèques d'E/S Java. Elles sont notoirement difficiles à comprendre au début. Mais s'ils voyaient simplement les classes comme un ensemble d'encapsuleurs autour d'un InputStream, la vie serait beaucoup plus facile.

Tête la première :Cela ne semble pas si mal. Vous êtes toujours un excellent modèle, et l'améliorer est juste une question d'éducation publique, n'est-ce pas ?

Décorateur:Il y a plus, j'en ai peur. J'ai des problèmes de frappe : vous voyez, les gens prennent parfois un morceau de code client qui repose sur des types spécifiques et introduisent des décorateurs sans réfléchir à tout. Maintenant, une chose formidable à propos de moi est que vous pouvez généralement insérer des décorateurs de manière transparente et le client n'a jamais besoin de savoir qu'il a affaire à un décorateur. Mais comme je l'ai dit, certains codes dépendent de types spécifiques et lorsque vous commencez à introduire des décorateurs, boum ! De mauvaises choses se produisent.

Tête la première :Eh bien, je pense que tout le monde comprend qu'il faut être prudent lors de l'insertion des décorateurs. Je ne pense pas que ce soit une raison pour être trop déprimé.

Décorateur:Je sais, j'essaie de ne pas l'être. J'ai aussi le problème que l'introduction de décorateurs peut augmenter la complexité du code nécessaire pour instancier le composant. Une fois que vous avez des décorateurs, vous devez non seulement instancier le composant, mais aussi l'envelopper avec on ne sait combien de décorateurs.

Tête la première :J'interviewerai les modèles Factory et Builder la semaine prochaine. J'ai entendu dire qu'ils peuvent être très utiles à ce sujet ?

Décorateur:C'est vrai, je devrais parler plus souvent à ces gars-là.

Tête la première :Eh bien, nous pensons tous que vous êtes un excellent modèle pour créer des designs flexibles et rester fidèle au principe ouvert-fermé, alors gardez la tête haute et pensez positivement !

Décorateur:Je ferai de mon mieux, merci.



Des outils pour votre boîte à outils de conception

Vous avez un autre chapitre à votre actif et un nouveau principe et modèle dans votre boîte à outils.

Principes OO

Notions de base sur OO

- Encapsuler ce qui varie.
- Privilégier la composition plutôt que l'héritage (éjecter l'héritance)
- Programmez vers des interfaces, pas vers des implémentations.
- Efforcez-vous d'obtenir des conceptions faiblement couplées entre les objets qui interagissent.
- Les cours devraient être ouverts pour prolongation mais fermés pour modification.

Abstraction
Encapsulation
Polymorphisme

Nous disposons désormais du principe d'ouverture-fermeture pour nous guider. Nous allons nous efforcer de concevoir notre système de manière à ce que les parties fermées soient isolées de nos nouvelles extensions.

Modèles OO

StratOp par exemple berm-euhdavetjelnes-adfeafminlyesofa aolgoer-itthom-s, de nombreux modèles existent. Ainsi, le patron de la délegation, qui varient en profondeur en fonction des composants, est un autre exemple de modèle OO.

interchwahneenarboelsnep.eonSosbirbjeailtcitgivsleattnosgtahesnesotalagtore, qui varient en profondeur en fonction des composants.

flexible auto-amltateircnaalitlyive à la sous-classe pour l'extension fonctionnalité.

Et voici notre premier modèle pour créer des designs qui respectent le principe ouvert-fermé. Ou était-ce vraiment le premier ? Existe-t-il un autre modèle que nous avons utilisé qui suit également ce principe ?



BULLET POINTS

f L'héritage est une forme d'extension, mais pas nécessairement le meilleur moyen d'obtenir de la flexibilité dans nos conceptions.

f Dans nos conceptions, nous devrions permettre l'extension du comportement sans avoir besoin de modifier le code existant.

f La composition et la délégation peuvent souvent être utilisées pour ajouter de nouveaux comportements lors de l'exécution.

f Le modèle Decorator fournit une alternative à la sous-classe pour étendre le comportement.

f Le modèle Decorator implique un ensemble de classes décoratrices utilisées pour envelopper des composants concrets.

f Les classes décoratrices reflètent le type des composants qu'elles décorent. (En fait, elles sont du même type que les composants qu'elles décorent, soit par héritage, soit par implémentation d'interface.)

f Les décorateurs modifient le comportement de leurs composants en ajoutant de nouvelles fonctionnalités avant et/ou après (ou même à la place) des appels de méthode au composant.

f Vous pouvez envelopper un composant avec n'importe quel nombre de décorateurs.

f Les décorateurs sont généralement transparents pour le client du composant, à moins que le client ne s'appuie sur le type concret du composant.

f Les décorateurs peuvent entraîner de nombreux petits objets dans notre conception, et leur surutilisation peut être complexe.



Écrivez les méthodes cost() pour les classes suivantes (le pseudo-Java est acceptable). Voici notre solution :

```

classe publique Boisson {

    // déclarer des variables d'instance pour milkCost, //
    soyCost, mochaCost et whipCost, et // des getters et
    setters pour milk, soy, mocha // et whip.

    coût double public() {

        double condimentCost = 0.0; si
        (hasMilk()) {
            condimentCost += milkCost;
        }
        si (hasSoy()) {
            condimentCost += sojaCost;
        }
        si (hasMocha()) {
            condimentCost += mokaCost;
        }
        si (hasWhip()) {
            condimentCost += whipCost;
        }
        retourner condimentCost;
    }
}

classe publique DarkRoast étend Beverage {

    public DarkRoast() {
        description = "Torréfaction foncée la plus excellente";
    }

    coût double public() {
        renvoie 1,99 + super.cost();
    }
}

```

Sharpen your pencil

Solution



Un nouveau barman arrive

« Latte double moka au soja avec crème fouettée »

- 1 Tout d'abord, nous appelons cost() sur le décorateur le plus externe, Whip.

- 2 Whip appelle cost() sur Mocha.

- 3 Mocha appelle cost() sur un autre Mocha.

- 4 Ensuite, Mocha appelle cost() sur Soy.

- 5 Dernier topping ! Soy appelle cost() sur HouseBlend.

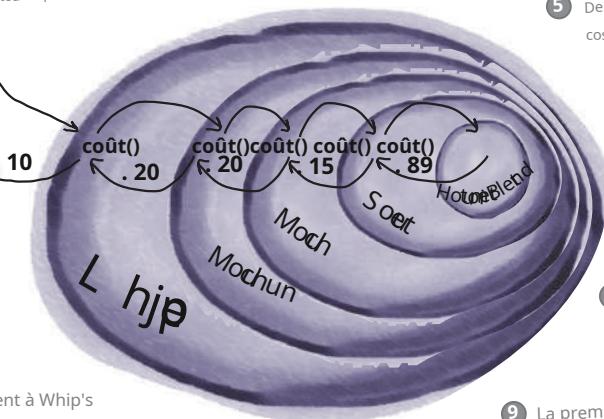
- 6 La méthode cost() de HouseBlend renvoie 0,89 et sort de la pile.

- 7 La méthode cost() de Soy ajoute 0,15 et renvoie le résultat, puis sort de la pile.

- 8 La deuxième méthode cost() de Mocha ajoute 0,20 et renvoie le résultat, puis sort de la pile.

- 9 La première méthode cost() de Mocha ajoute 0,20 et renvoie le résultat, puis sort de la pile.

- 10 Finalement, le résultat revient à Whip's cost(), qui ajoute 0,10, ce qui nous donne un coût final de 1,54 \$.





Sharpen your pencil Solution

Nos amis de Starbuzz ont introduit des tailles dans leur menu. Vous pouvez désormais commander un café dans les tailles tall, grande et venti (traduction : petit, moyen et grand). Starbuzz a vu cela comme une partie intrinsèque de la classe coffee, ils ont donc ajouté deux méthodes à la classe Beverage : setSize() et getSize(). Ils aimeraient également que les condiments soient facturés en fonction de la taille, ainsi, par exemple, le soja coûte respectivement 10¢, 15¢ et 20¢ pour les cafés tall, grande et venti.

Comment modifieriez-vous les classes de décorateur pour gérer ce changement d'exigences ? Voici notre solution.

```
classe abstraite publique CondimentDecorator étend Beverage {
```

```
    boisson publique;
```

```
    chaîne publique abstraite getDescription();
```

```
    Taille publique getSize() {
```

```
        retourner boisson.getSize();
```

```
    }
```

```
}
```

Nous avons ajouté une méthode, getSize(), pour les décorateurs qui renvoie simplement la taille de la boisson.



```
classe publique Soy étend CondimentDecorator {
```

```
    boisson publique soja (boisson) {
```

```
        cette.boisson = boisson;
```

```
}
```

```
    chaîne publique getDescription() {
```

```
        retourner boisson.getDescription() + ", Soja";
```

```
}
```

```
    coût double public() {
```

```
        coût double = boisson.coût();
```

```
        si (boisson.getSize() == Taille.TALL) {
```

```
            coût += .10;
```

```
        } else if (beverage.getSize() == Taille.GRANDE) {
```

```
            coût += .15;
```

```
        } sinon si (boisson.getSize() == Taille.VENTI) {
```

```
            coût += .20;
```

```
        }
```

```
        frais de retour;
```

```
}
```

```
}
```

Ici, nous obtenons la taille (qui se propage jusqu'à la boisson concrète) et ajoutons ensuite le coût approprié.



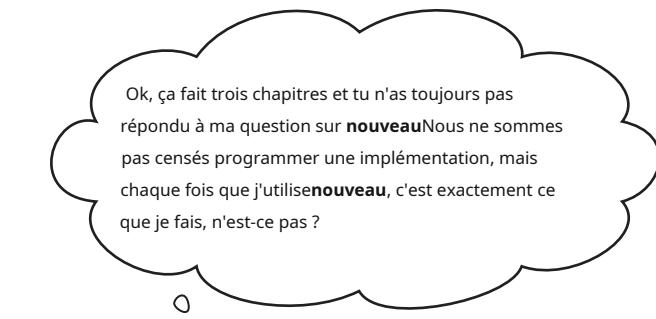
4^e modèle d'usine



Cuisiner avec les bienfaits d'OO



Préparez-vous à créer des designs OO faiblement couplés. Il y a plus pour fabriquer des objets plutôt que simplement utiliser les **nouveau** opérateur. Vous apprendrez que l'instanciation est une activité qui ne doit pas toujours être effectuée en public et peut souvent conduire à *problèmes de couplage*. Et nous ne voulons *pasque*, et nous ? Découvrez comment les Factory Patterns peuvent vous aider à éviter des dépendances embarrassantes.



Quand vous voyez « nouveau », pensez « béton ».

Oui, lorsque vous utilisez le **nouveau** opérateur vous instanciez certainement une classe concrète, il s'agit donc bien d'une implémentation et non d'une interface. Et vous faites une bonne observation : lier votre code à une classe concrète peut le rendre plus fragile et moins flexible.

Canard canard = nouveau MallardDuck();

Nous souhaitons utiliser des types abstraits pour garder le code flexible.

Mais nous devons créer une instance d'une classe concrète !

Lorsque nous disposons d'un ensemble complet de classes concrètes liées, nous finissons souvent par écrire du code comme celui-ci :

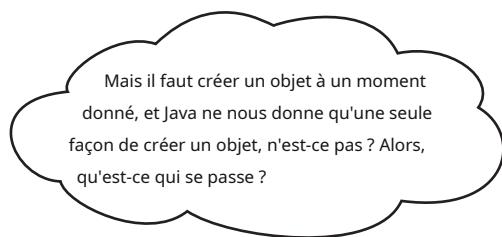
Canard canard;

```
si (pique-nique) {  
    canard = nouveau  
    MallardDuck(); } else if (chasse) {  
    canard = nouveau DecoyDuck(); }  
sinon si (dans la baignoire) {  
    canard = nouveau RubberDuck();  
}
```

Nous avons un tas de classes de canards différentes, et nous ne savons pas avant l'exécution laquelle nous devons instancier.

Ici, nous avons plusieurs classes concrètes en cours dinstanciation, et la décision de laquelle instancier est prise au moment de l'exécution en fonction d'un ensemble de conditions.

Lorsque vous voyez un code comme celui-ci, vous savez que lorsque viendra le temps de procéder à des modifications ou des extensions, vous devrez rouvrir ce code et examiner ce qui doit être ajouté (ou supprimé). Souvent, ce type de code se retrouve dans plusieurs parties de l'application, ce qui rend la maintenance et les mises à jour plus difficiles et plus sujettes aux erreurs.



Quel est le problème avec « nouveau » ?

Techniquement, il n'y a rien de mal avec `lenouveauopérateur`. Après tout, c'est un élément fondamental de la plupart des langages orientés objet modernes. Le véritable coupable est notre vieil ami CHANGE et la façon dont le changement affecte notre utilisation de `nouveau`.

En codant pour une interface, vous savez que vous pouvez vous protéger de nombreux changements qui pourraient survenir dans un système à l'avenir. Pourquoi ? Si votre code est écrit pour une interface, il fonctionnera avec toutes les nouvelles classes implémentant cette interface grâce au polymorphisme. Cependant, lorsque vous avez du code qui utilise de nombreuses classes concrètes, vous cherchez des problèmes car ce code peut devoir être modifié à mesure que de nouvelles classes concrètes sont ajoutées. En d'autres termes, votre code ne sera pas « fermé à la modification ». Pour étendre votre code avec de nouveaux types concrets, vous devrez le rouvrir.

Alors, que pouvez-vous faire ? C'est dans des moments comme ceux-ci que vous pouvez vous appuyer sur les principes de conception OO pour chercher des indices. N'oubliez pas que notre premier principe traite du changement et nous guide vers identifier les aspects qui varient et les séparer de ce qui reste le même.

N'oubliez pas que les dessins devrait être « ouvert à la prolongation mais fermé pour modification. » Voir Chapitre 3 pour une révision.



Comment pourriez-vous prendre toutes les parties de votre application qui instantient des classes concrètes et les séparer ou les encapsuler du reste de votre application ?

Identifier les aspects qui varient

Disons que vous avez une pizzeria et qu'en tant que propriétaire d'une pizzeria de pointe à Objectville, vous pourriez finir par écrire du code comme celui-ci :

```
Commande de pizzaPizza() {  
    Pizza pizza = nouvelle Pizza();  
  
    pizza.préparer();  
    pizza.cuire();  
    pizza.couper();  
    boîte à pizza();  
    retourner la pizza;  
}
```



Pour plus de flexibilité, nous voulons vraiment que ce soit une classe ou une interface abstraite, mais malheureusement, nous ne pouvons pas instancier directement l'une ou l'autre.

Mais il vous faut plus d'un type de pizza...

Vous ajouteriez alors du code qui détermine le type de pizza approprié et ensuite se passe fabrication la pizza:

```
Commande de pizzaPizza( Type de chaîne ) {  
    Pizza pizza;  
  
    si (type.equals("fromage")) {  
        pizza = nouvelle CheesePizza(); } else  
    if (type.equals("grec")) {  
        pizza = nouvelle GreekPizza(); }  
    } sinon si (type.equals("pepperoni")) {  
        pizza = nouvelle PepperoniPizza(); }  
  
    pizza.préparer();  
    pizza.cuire();  
    pizza.couper();  
    boîte à pizza();  
    retourner la pizza;  
}
```

Nous passons maintenant au type de pizza à commanderPizza.

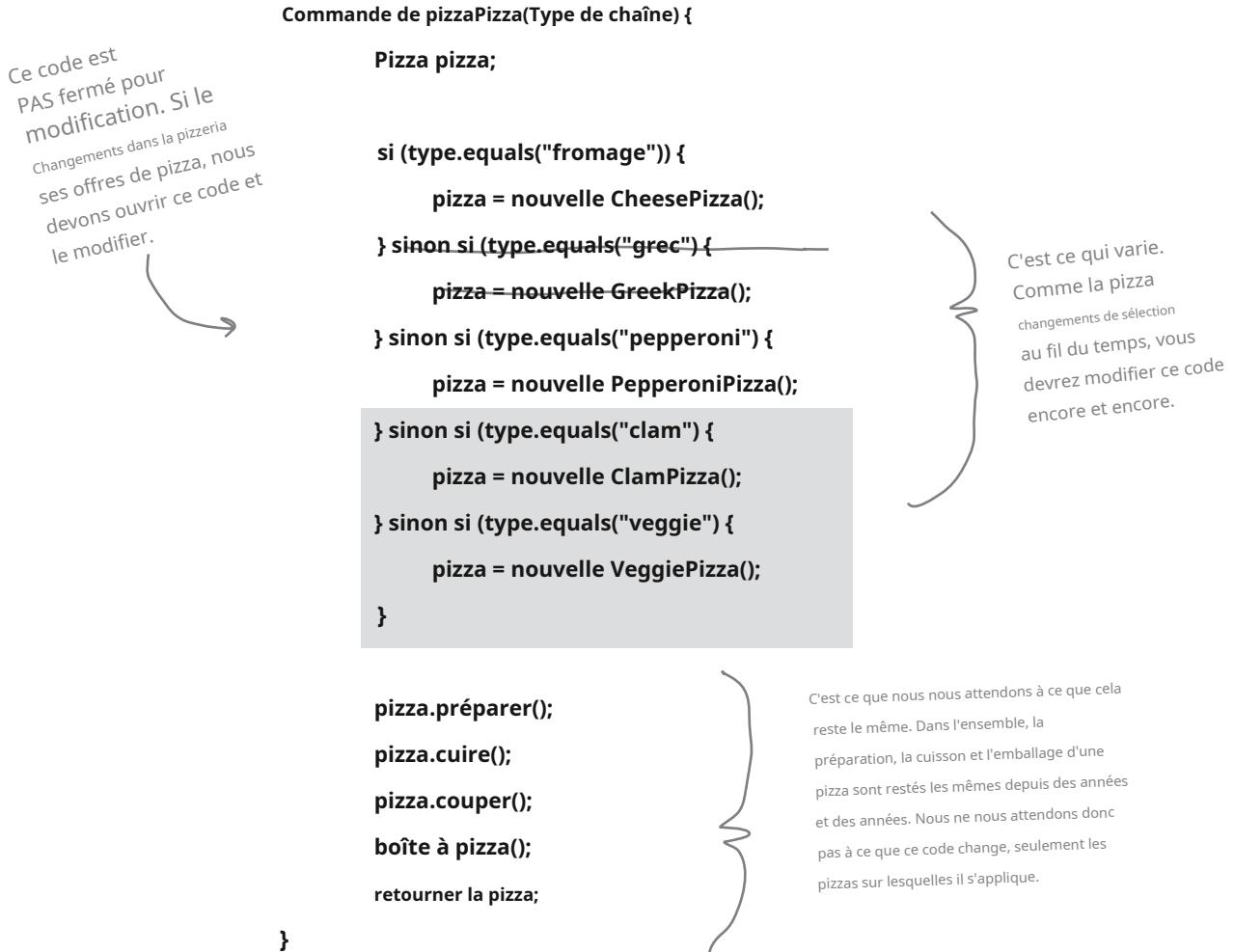
En fonction du type de pizza, nous instancions la classe concrète appropriée et l'assignons à la variable d'instance pizza. Notez que chaque pizza ici doit implémenter l'interface Pizza.

Une fois que nous avons une pizza, nous la préparons (vous savez, nous étalons la pâte, mettons la sauce et ajoutons les garnitures), puis nous la faisons cuire, la coupons et la mettons en boîte !

Chaque sous-type de pizza (pizza au fromage, pizza grecque, etc.) sait se préparer.

Mais la pression est forte pour ajouter plus de types de pizzas

Vous vous rendez compte que tous vos concurrents ont ajouté quelques pizzas tendance à leur menu : la pizza aux palourdes et la pizza végétarienne. Évidemment, vous devez suivre la concurrence, vous allez donc ajouter ces éléments à votre menu. Et comme vous n'avez pas beaucoup vendu de pizzas grecques ces derniers temps, vous décidez de les retirer du menu :



De toute évidence, traiter avec *lequel* la classe concrète instanciée perturbe vraiment notre méthode *orderPizza()* et l'empêche d'être fermée pour modification. Mais maintenant que nous savons ce qui varie et ce qui ne varie pas, il est probablement temps de l'encapsuler.

Création d'objets encapsulés

Nous savons donc maintenant qu'il serait préférable de déplacer la création d'objet hors de la méthode `orderPizza()`. Mais comment ? Eh bien, ce que nous allons faire, c'est prendre le code de création et le déplacer vers un autre objet qui ne s'occupera que de la création de pizzas.

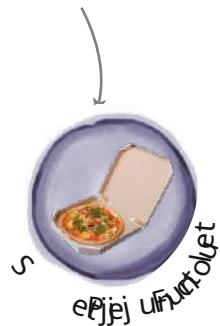
```
Commande de pizzaPizza(Type de chaîne) {  
    Pizza pizza;  
  
    pizza.préparer();  
    pizza.cuire();  
    pizza.couper();  
    boîte à pizza();  
    retourner la pizza;  
}
```

Qu'est-ce qui va se passer ici ?

Nous commençons par extraire
le code de création d'objet de la
méthode `orderPizza()`.

```
si (type.equals("fromage")) {  
    pizza = nouvelle CheesePizza();  
} sinon si (type.equals("pepperoni")) {  
    pizza = nouvelle PepperoniPizza();  
} sinon si (type.equals("clam")) {  
    pizza = nouvelle ClamPizza();  
} sinon si (type.equals("veggie")) {  
    pizza = nouvelle VeggiePizza();  
}
```

Ensuite, nous plaçons ce code dans un objet qui
se chargera uniquement de créer des pizzas. Si
un autre objet a besoin de créer une pizza, c'est
à cet objet qu'il faut s'adresser.



**Nous avons un nom pour ce nouvel objet : nous
l'appelons une Usine.**

Les usines gèrent les détails de la création d'objets. Une fois que nous avons une `SimplePizzaFactory`, notre méthode `orderPizza()` devient un client de cet objet. Chaque fois qu'elle a besoin d'une pizza, elle demande à l'usine à pizza d'en fabriquer une. Fini le temps où la méthode `orderPizza()` avait besoin de savoir faire la différence entre les pizzas grecques et les pizzas aux palourdes. Désormais, la méthode `orderPizza()` se soucie simplement d'obtenir une pizza qui implémente l'interface `Pizza` afin de pouvoir appeler `prepare()`, `bake()`, `cut()` et `box()`.

Il nous reste encore quelques détails à régler ici ; par exemple, par quoi la méthode `orderPizza()` remplace-t-elle son code de création ? Implémentons une simple usine pour la pizzeria et découvrons...

Construire une usine à pizza simple

Nous allons commencer par l'usine elle-même. Ce que nous allons faire, c'est définir une classe qui encapsule la création d'objets pour toutes les pizzas. La voici...

Voici notre nouvelle classe, la SimplePizzaFactory. Elle a une seule mission : créer des pizzas pour ses clients.

classe publique SimplePizzaFactory {

```
public Pizza createPizza (type de chaîne) {
    Pizza pizza = null;
```

```
    si (type.equals("fromage")) {
        pizza = nouvelle CheesePizza();
    } sinon si (type.equals("pepperoni")) {
        pizza = nouvelle PepperoniPizza(); }
    else if (type.equals("clam")) {
        pizza = nouvelle ClamPizza();
    } sinon si (type.equals("veggie")) {
        pizza = nouvelle VeggiePizza();
    }
    retourner la pizza;
}
```

Nous définissons d'abord une méthode createPizza() dans l'usine. Il s'agit de la méthode que tous les clients utiliseront pour instancier de nouveaux objets.

Voici le code que nous avons extrait de la méthode orderPizza().

Ce code est toujours paramétré par le type de pizza, tout comme notre méthode orderPizza() d'origine.

there are no
Dumb Questions

Q:

Quel est l'avantage de cela ? On dirait que nous ne faisons que déplacer le problème vers un autre objet.

UN:

Il faut se rappeler que SimplePizzaFactory peut avoir de nombreux clients. Nous n'avons vu que la méthode orderPizza() ; cependant, il peut y avoir une classe PizzaShopMenu qui utilise l'usine pour obtenir des pizzas correspondant à leur description et à leur prix actuels. Nous pouvons également avoir une classe HomeDelivery qui gère les pizzas d'une manière différente de notre classe PizzaShop mais qui est également un client de l'usine.

Ainsi, en encapsulant la création de pizza dans une seule classe, nous n'avons désormais qu'un seul endroit pour apporter des modifications lorsque l'implémentation change.

Et n'oubliez pas que nous sommes également sur le point de supprimer les instantiations concrètes de notre code client.

Q:

J'ai vu une conception similaire où une usine comme celle-ci est définie comme une méthode statique. Quelle est la différence ?

UN:

Définir une fabrique simple comme méthode statique est une technique courante et est souvent appelée fabrique statique. Pourquoi utiliser une méthode statique ? Parce que vous n'avez pas besoin d'instancier un objet pour utiliser la méthode create. Mais elle présente également l'inconvénient de ne pas pouvoir sous-classer et modifier le comportement de la méthode create.

Retravailler la classe PizzaStore

Il est maintenant temps de corriger notre code client. Ce que nous voulons faire, c'est compter sur l'usine pour créer les pizzas à notre place. Voici les changements :

```
classe publique PizzaStore {  
    Usine SimplePizzaFactory ;  
  
    PizzaStore public (usine SimplePizzaFactory) {  
        cette.usine = usine;  
    }  
  
    commande de pizza publiquePizza (type de chaîne) {  
        Pizza pizza;  
  
        pizza = usine.createPizza(type);  
  
        pizza.préparer();  
        pizza.cuire();  
        pizza.couper();  
        boîte à pizza();  
  
        retourner la pizza;  
    }  
  
    // autres méthodes ici  
}
```

Nous donnons d'abord à PizzaStore une référence à une SimplePizzaFactory.

PizzaStore obtient l'usine qui lui est transmise dans le constructeur.

Notez que nous avons remplacé l'opérateur new par une méthode createPizza dans l'objet factory. Plus d'instanciations concrètes ici !

Et la méthode orderPizza() utilise l'usine pour créer ses pizzas en passant simplement le type de la commande.



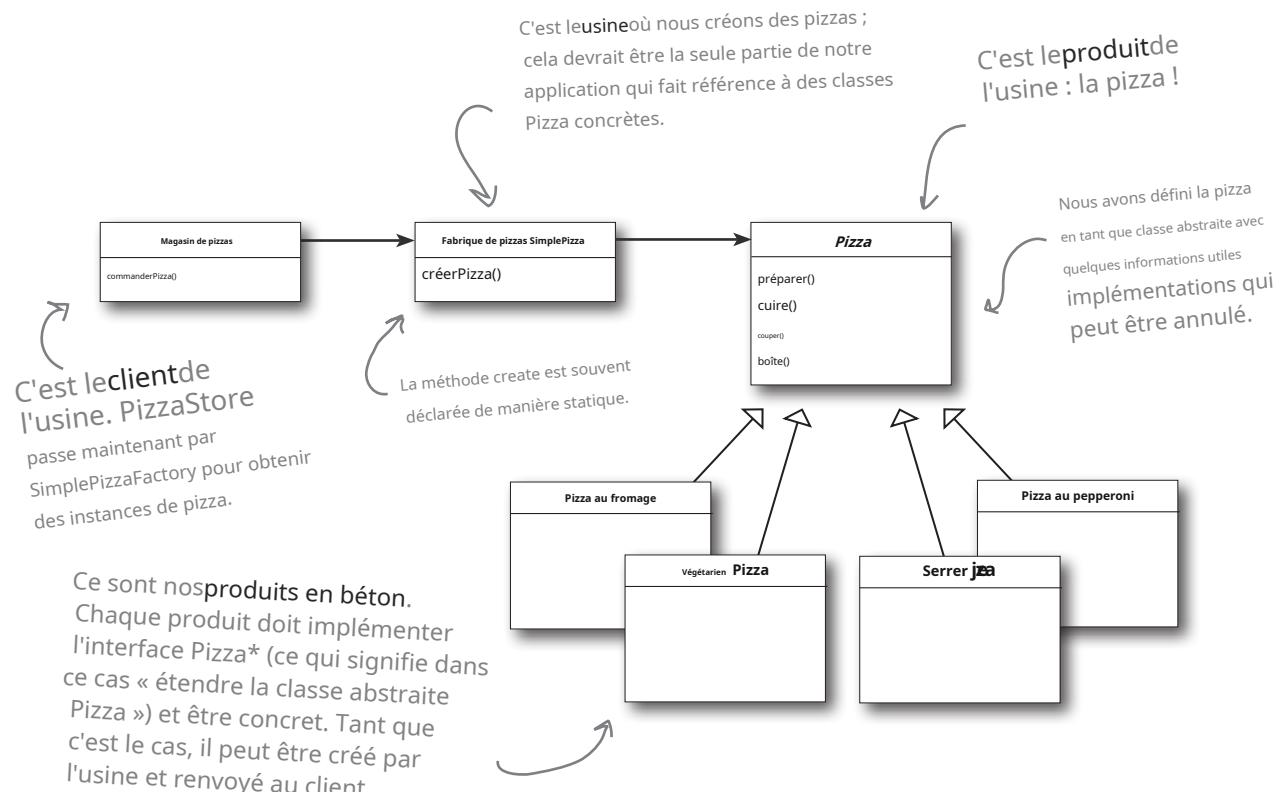
Nous savons que la composition d'objets nous permet de modifier le comportement de manière dynamique au moment de l'exécution (entre autres choses) car nous pouvons échanger des implémentations. Comment pourrions-nous utiliser cela dans PizzaStore ? Quelles implémentations d'usine pourrions-nous échanger ?

(n'oublions pas New Haven aussi).
Nous ne savons pas pour vous, mais nous pensons aux usines à pizza de style New York, Chicago et Californie

Définition de la Simple Factory

Le Simple Factory n'est pas réellement un Design Pattern, c'est plutôt un idiomme de programmation. Mais il est couramment utilisé, nous lui décernons donc une mention honorable pour le modèle Head First. Certains développeurs confondent cet idiomme avec le Factory Pattern, mais la prochaine fois que cela se produit, vous pouvez subtilement montrer que vous connaissez votre sujet ; ne vous pavanez simplement pas en leur expliquant la distinction.

Ce n'est pas parce que Simple Factory n'est pas un VRAI modèle que nous ne devons pas vérifier comment il est conçu. Jetons un œil au diagramme de classes de notre nouvelle pizzeria :



Considérez Simple Factory comme un échauffement. Ensuite, nous allons explorer deux modèles très résistants qui sont tous deux des usines. Mais ne vous inquiétez pas, il y a encore plus de pizza à venir !

* Juste un autre rappel : dans les modèles de conception, l'expression « implémenter une interface » ne signifie PAS toujours « écrire une classe qui implémente une interface Java, en utilisant le mot-clé « implements » dans la déclaration de classe ». Dans l'utilisation générale de l'expression, une classe concrète implémentant une méthode d'un supertype (qui peut être une classe abstraite OU une interface) est toujours considérée comme « implémentant l'interface » de ce supertype.

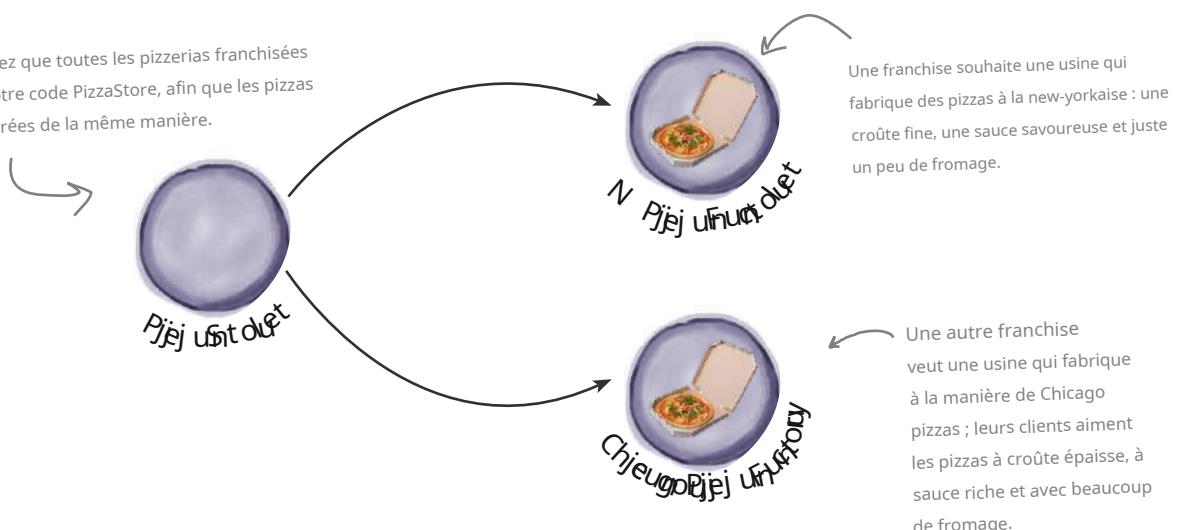
Franchise de pizzeria

Votre pizzeria Objectville a connu un tel succès que vous avez surclassé la concurrence et que désormais tout le monde veut une pizzeria dans son quartier. En tant que franchiseur, vous souhaitez garantir la qualité des opérations de franchise et vous souhaitez donc qu'ils utilisent votre code éprouvé.

Mais qu'en est-il des différences régionales ? Chaque franchise peut vouloir proposer des styles de pizzas différents (New York, Chicago et Californie, pour n'en citer que quelques-uns), en fonction de l'emplacement du magasin franchisé et des goûts des connaisseurs de pizza locaux.

Oui, différentes régions des États-Unis servent des styles de pizza très différents : des pizzas à croûte épaisse de Chicago, à la pizza à croûte fine de New York, en passant par la pizza ressemblant à des craquelins de Californie (certains diraient garnie de fruits et de noix).

Vous souhaitez que toutes les pizzerias franchisées exploitent votre code PizzaStore, afin que les pizzas soient préparées de la même manière.



Nous avons vu une approche...

Si nous retirons SimplePizzaFactory et créons trois usines différentes (NYPizzaFactory, ChicagoPizzaFactory et CaliforniaPizzaFactory), nous pouvons alors simplement composer le PizzaStore avec l'usine appropriée et la franchise est prête à être lancée. C'est une approche possible.

Voyons à quoi cela ressemblerait...

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.orderPizza("Végétarien");
```

Ici, nous créons une usine pour faire des pizzas de style new-yorkais.

Ensuite, nous créons un PizzaStore et lui transmettons une référence à l'usine NY.

... et quand on fait des pizzas, on obtient des pizzas à la new-yorkaise.

```
ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory(); PizzaStore
chicagoStore = new PizzaStore(chicagoFactory);
chicagoStore.orderPizza("Végétarien");
```

De même pour les pizzerias de Chicago : nous créons une usine pour les pizzas de Chicago et nous créons un magasin qui est composé d'une usine de Chicago. Lorsque nous faisons des pizzas, nous obtenons celles de style Chicago.

Mais vous aimerez un peu plus de contrôle qualité...

Vous avez donc testé l'idée de SimpleFactory et vous avez découvert que les franchises utilisaient votre usine pour créer des pizzas, mais commençaient à employer leurs propres procédures locales pour le reste du processus : elles cuisaient les choses un peu différemment, elles oubblaient de couper la pizza et elles utilisaient des boîtes tierces.

En repensant un peu au problème, vous voyez que ce que vous aimerez vraiment faire, c'est créer un cadre qui lie le magasin et la création de pizzas, tout en permettant aux choses de rester flexibles.

Dans notre code initial, avant SimplePizzaFactory, nous avions le code de fabrication de pizza lié au PizzaStore, mais il n'était pas flexible. Alors, comment pouvons-nous avoir notre pizza et la manger aussi ?

Je fais des pizzas depuis des années, alors j'ai pensé ajouter mes propres « améliorations » aux procédures de PizzaStore...



Ce n'est pas ce que vous recherchez dans une bonne franchise. Vous ne voulez PAS savoir ce qu'il met sur ses pizzas.

Un cadre pour la pizzeria

La est un moyen de localiser toutes les activités de fabrication de pizza dans la classe PizzaStore, et de donner aux franchises la liberté d'avoir leur propre style régional.

Ce que nous allons faire est de remettre la méthode createPizza() dans PizzaStore, mais cette fois en tant que méthode abstraite, puis de créer une sous-classe PizzaStore pour chaque style régional.

Commençons par examiner les changements apportés au PizzaStore :

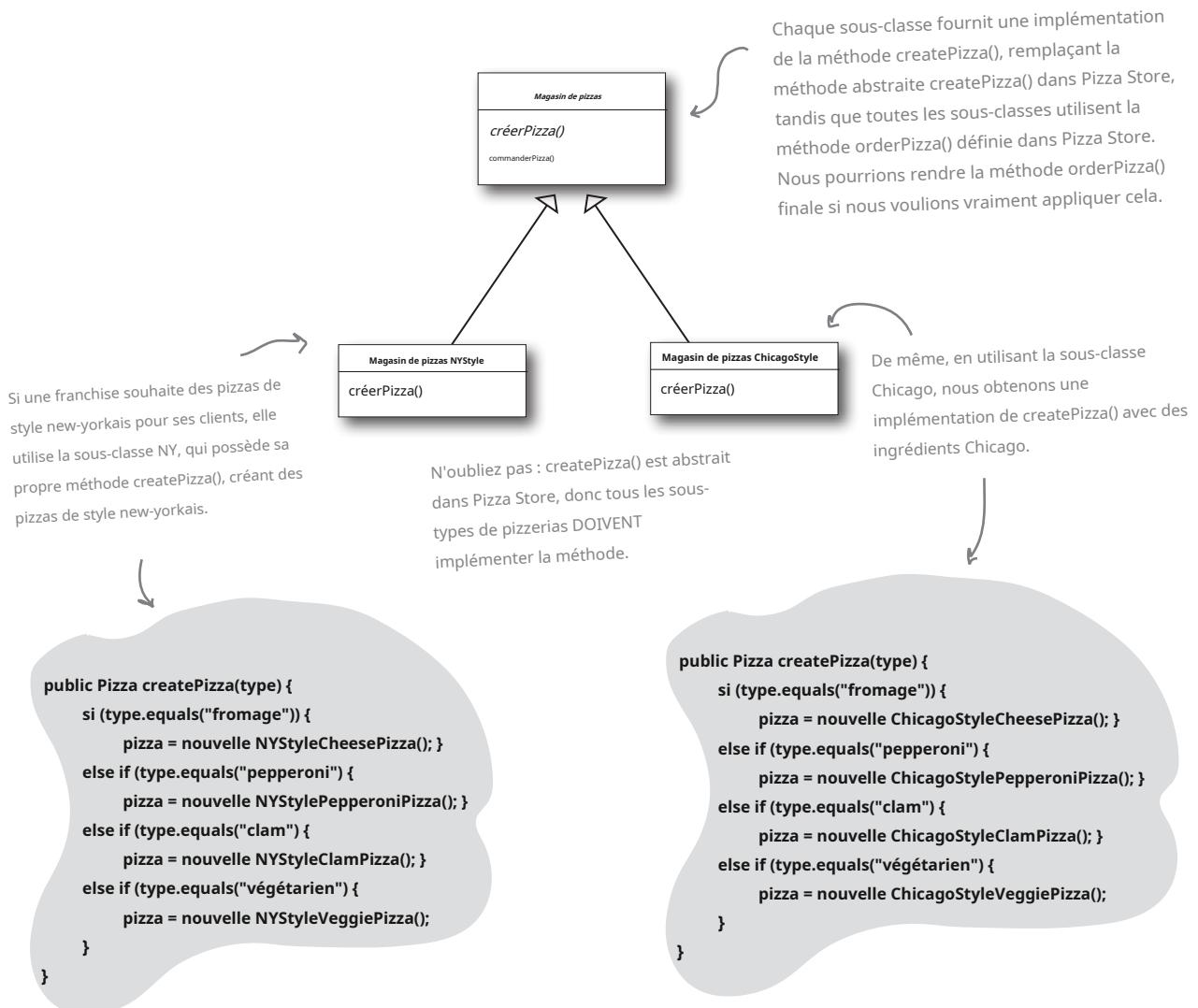
```
PizzaStore est désormais abstrait (voir pourquoi ci-dessous).  
↓  
classe abstraite publique PizzaStore {  
  
commande de pizza publiquePizza (type de chaîne) {  
    Pizza pizza;  
        pizza = créerPizza(type); ← Maintenant, createPizza redevient un  
        pizza.préparer();          appel à une méthode dans PizzaStore  
        pizza.cuire();           plutôt qu'à un objet d'usine.  
        pizza.couper();  
        boîte à pizza();         ← Tout cela se ressemble...  
  
        retourner la pizza;  
}  
  
abstrait Pizza createPizza(String type); ← Nous avons maintenant déplacé notre objet  
}  
        Notre « méthode d'usine » est  
        désormais abstraite dans PizzaStore.
```

Nous avons maintenant un magasin en attente de sous-classes ; nous allons avoir une sous-classe pour chaque type régional (NYPizzaStore, ChicagoPizzaStore, CaliforniaPizzaStore) et chaque sous-classe va prendre la décision sur ce qui constitue une pizza. Voyons comment cela va fonctionner.

Permettre aux sous-classes de décider

N'oubliez pas que la pizzeria dispose déjà d'un système de commande bien rodé dans la méthode `orderPizza()` et que vous souhaitez vous assurer qu'il est cohérent dans toutes les franchises.

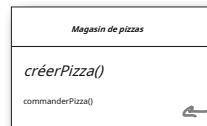
Ce qui varie selon les pizzerias régionales, c'est le style des pizzas qu'elles préparent (la pizza de New York a une croûte fine, celle de Chicago a une croûte épaisse, etc.). Nous allons donc intégrer toutes ces variantes dans la méthode `createPizza()` et la rendre responsable de la création du bon type de pizza. Pour ce faire, nous laissons chaque sous-classe de `Pizza Store` définir à quoi ressemble la méthode `createPizza()`. Nous aurons donc un certain nombre de sous-classes concrètes de `Pizza Store`, chacune avec ses propres variantes de pizza, toutes adaptées au cadre de `Pizza Store` et utilisant toujours la méthode `orderPizza()` bien réglée.





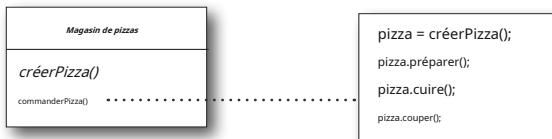
Je ne comprends pas. Les sous-classes de PizzaStore ne sont que des sous-classes. Comment décident-elles quoi que ce soit ? Je ne vois aucun code de prise de décision logique dans NYStylePizzaStore....

Eh bien, pensez-y du point de vue de la méthode orderPizza() de PizzaStore : elle est définie dans le PizzaStore abstrait, mais les types concrets ne sont créés que dans les sous-classes.



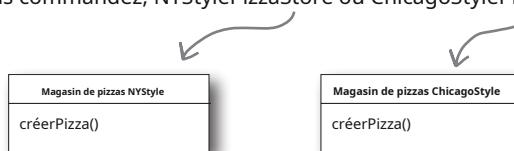
orderPizza() est défini dans la classe abstraite PizzaStore, et non dans les sous-classes. La méthode n'a donc aucune idée de la sous-classe qui exécute réellement le code et prépare les pizzas.

Maintenant, pour aller un peu plus loin, la méthode orderPizza() fait beaucoup de choses avec un objet Pizza (comme préparer, cuire, couper, mettre en boîte), mais comme Pizza est abstrait, orderPizza() n'a aucune idée des classes concrètes réelles impliquées. En d'autres termes, il est découplé !



orderPizza() appelle createPizza() pour obtenir un objet pizza. Mais quel type de pizza va-t-il obtenir ? La méthode orderPizza() ne peut pas décider ; elle ne sait pas comment. Alors qui décide ?

Lorsque orderPizza() appelle createPizza(), l'une de vos sous-classes sera appelée à l'action pour créer une pizza. Quel type de pizza sera préparé ? Eh bien, cela dépend du choix de la pizzeria où vous commandez, NYStylePizzaStore ou ChicagoStylePizzaStore.



Donc, les sous-classes prennent-elles une décision en temps réel ? Non, mais du point de vue de orderPizza(), si vous choisissez un NYStylePizzaStore, cette sous-classe détermine quelle pizza est préparée. Les sous-classes ne « décident » donc pas vraiment, c'est toi qui a décidé en choisissant le magasin que vous vouliez, mais ils déterminent également quel type de pizza est préparé.

Créons une pizzeria

Être une franchise a ses avantages. Vous bénéficiez de toutes les fonctionnalités de PizzaStore gratuitement. Tout ce que les magasins régionaux doivent faire est de sous-classer PizzaStore et de fournir une méthode createPizza() qui implémente leur style de pizza. Nous nous occuperons des trois principaux styles de pizza pour les franchisés.

Voici le style régional de New York :

```
createPizza() renvoie une Pizza, et la
sous-classe est entièrement responsable
de la Pizza concrète qu'elle instancie.
```

```
classe publique NYPizzaStore étend PizzaStore {
```

```
Pizza createPizza(élément de chaîne) {
    si (élément.equals("fromage")) {
        renvoyer une nouvelle NYStyleCheesePizza(); }
    else if (item.equals("veggie")) {
        renvoyer une nouvelle NYStyleVeggiePizza(); }
    else if (item.equals("clam")) {
        renvoyer un nouveau NYStyleClamPizza(); }
    else if (item.equals("pepperoni")) {
        renvoyer un nouveau
        NYStylePepperoniPizza(); } sinon renvoyer null;
}
```

```
Nous devons implémenter
createPizza(), car il est
abstrait dans PizzaStore.
```

```
C'est ici que nous créons nos classes
concrètes. Pour chaque type de pizza,
nous créons le style NY.
```

* Notez que la méthode orderPizza() de la superclasse n'a aucune idée de la pizza que nous créons ; elle sait juste qu'elle peut la préparer, la cuire, la couper et la mettre en boîte !

Une fois que nous aurons créé nos sous-classes PizzaStore, il sera temps de passer à la commande d'une ou deux pizzas. Mais avant cela, pourquoi ne pas essayer de créer les pizzerias de style Chicago et Californien de la page suivante ?



Sharpen your pencil

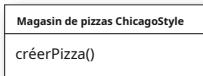
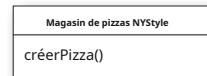
Nous avons terminé le NYPizzaStore ; il n'en reste plus que deux et nous serons prêts à franchiser ! Écrivez les implémentations de PizzaStore de style Chicago et de style Californien ici :

Déclaration d'une méthode de fabrique

Avec seulement quelques transformations apportées à la classe PizzaStore, nous sommes passés d'un objet gérant l'instanciation de nos classes concrètes à un ensemble de sous-classes qui assument désormais cette responsabilité. Regardons cela de plus près :

```
classe abstraite publique PizzaStore {
    commande de pizza publiquePizza (type de chaîne) {
        Pizza pizza;
        pizza = createPizza(type);
        pizza.préparer();
        pizza.cuire();
        pizza.couper();
        boîte à pizza();
        retourner la pizza;
    }
    protégé abstrait Pizza createPizza(String type);
    // autres méthodes ici
}
```

Les sous-classes de PizzaStore gère l'instanciation de l'objet pour nous dans la méthode createPizza().



Toute la responsabilité de l'instanciation des pizzas a été transférée vers une méthode qui agit comme une usine.

Le code de près

Une méthode d'usine gère la création d'objets et l'encapsule dans une sous-classe. Cela dissocie le code client dans la superclasse du code de création d'objets dans la sous-classe.

Produit abstrait factoryMethod(type String)

Une méthode d'usine est abstrait donc les sous-classes sont comptées pour gérer la création d'objets.

Une méthode d'usine renvoie un produit qui est généralement utilisé dans les méthodes définies dans la superclasse.

Une méthode d'usine peut être paramétrée (ou pas) choisir parmi plusieurs variantes d'un produit.
Une méthode d'usine isole le client (le code) dans la superclasse, comme orderPizza()) de savoir quel type de produit concret est réellement créé.

Voyons comment ça marche : commander des pizzas avec la méthode de la fabrique à pizza



Alors comment commandent-ils ?

- ➊ Tout d'abord, Joel et Ethan ont besoin d'une instance de PizzaStore. Joel doit créer une instance de ChicagoPizzaStore et Ethan a besoin d'une instance de NYPizzaStore.
- ➋ Avec un PizzaStore en main, Ethan et Joel appellent tous deux la méthode orderPizza() et transmettent le type de pizza qu'ils souhaitent (fromage, végétarien, etc.).
- ➌ Pour créer les pizzas, la méthode createPizza() est appelée, qui est définie dans les deux sous-classes NYPizzaStore et ChicagoPizzaStore. Comme nous les avons définies, NYPizzaStore instancie une pizza de style NY et ChicagoPizzaStore instancie une pizza de style Chicago. Dans les deux cas, la pizza est renvoyée à la méthode orderPizza().
- ➍ La méthode orderPizza() n'a aucune idée du type de pizza qui a été créée, mais elle sait que c'est une pizza et la prépare, la cuite, la coupe et la met en boîte pour Ethan et Joel.

Voyons comment ces pizzas sont réellement préparées sur commande...

Derrière les scènes



1

Suivons la commande d'Ethan : nous avons d'abord besoin d'un NY PizzaStore :

Pizzeria nyPizzaStore = new NY PizzaStore();

Crée une instance de NY PizzaStore.

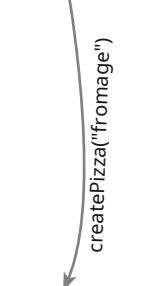


2

Maintenant que nous avons un magasin, nous pouvons prendre une commande :

nyPizzaStore.orderPizza("fromage");

La méthode orderPizza() est appelée sur l'instance nyPizzaStore (la méthode définie dans PizzaStore s'exécute).



3

La méthode orderPizza() appelle ensuite la méthode createPizza() :

Pizza pizza = createPizza("fromage");

N'oubliez pas que createPizza(), la méthode factory, est implémentée dans la sous-classe. Dans ce cas, elle renvoie une pizza au fromage de style new-yorkais.



4

Finalement, nous avons la pizza non préparée en main et la méthode orderPizza() termine sa préparation :

**pizza.préparer();
pizza.cuire();
pizza.couper();
boîte à pizza();**

La méthode orderPizza() récupère une Pizza, sans savoir exactement de quelle classe concrète il s'agit.

Toutes ces méthodes sont définies dans la pizza spécifique renvoyée par la méthode d'usine createPizza(), définie dans NY PizzaStore.

Il nous manque juste un ejefr: Des pizzas !

Notre pizzeria ne serait pas très populaire sans quelques pizzas, alors mettons-les en œuvre



Nous commencerons avec une classe Pizza abstraite, et toutes les pizzas concrètes en découlent.

classe abstraite publique Pizza {

```
Nom de la chaîne;  
Pâte à ficelle;  
Sauce à ficelle;  
Liste<String> garnitures = new ArrayList<String>();
```

```
void préparer() {  
    System.out.println("Préparation de " + nom);  
    System.out.println("Lancement de la pâte...");  
    System.out.println("Ajout de sauce...");  
    System.out.println("Ajout de garnitures : ");  
    for (String topping : toppings) {  
        System.out.println(" " + garniture);  
    }  
}
```

```
void cuire() {  
    System.out.println("Cuire au four pendant 25 minutes à 350");  
}
```

```
void couper() {  
    System.out.println("Découper la pizza en tranches diagonales");  
}
```

```
boîte vide() {  
    System.out.println("Placez la pizza dans la boîte officielle PizzaStore");  
}
```

```
chaîne publique getName() {  
    renvoyer le nom;  
}  
}
```

Chaque pizza a un nom, un type de pâte, un type de sauce et un ensemble de garnitures.

La préparation suit une
nombre d'étapes dans
une séquence particulière.

La classe abstraite fournit quelques
valeurs par défaut de base pour la
cuisson, la découpe et la mise en boîte.

N'OUBLIEZ PAS : nous ne fournissons pas d'instructions d'importation et de package dans les listes de codes. Obtenez le code source complet sur le site Web de wickedlysmart à l'adresse <https://wickedlysmart.com/head-first-design-patterns>

Si vous perdez cette URL, vous pouvez toujours la retrouver rapidement dans la section Intro.

Il ne nous manque plus que quelques sous-classes concrètes... que diriez-vous de définir des pizzas au fromage de style New York et Chicago ?

```
classe publique NYStyleCheesePizza étend Pizza {  
  
    public NYStyleCheesePizza() {  
        nom = "Pizza à la sauce et au fromage de style new-  
        yorkais"; pâte = "Pâte à croûte mince";  
        sauce = "Sauce Marinara";  
  
        toppings.add("Fromage Reggiano râpé");  
    }  
}
```

La pizza NY a sa propre sauce de style marinara et sa croûte fine.



Et une garniture, du fromage Reggiano !

```
classe publique ChicagoStyleCheesePizza étend Pizza {  
  
    public ChicagoStyleCheesePizza() {  
        name = "Pizza au fromage à croûte épaisse de style Chicago";  
        dough = "Pâte à croûte extra épaisse";  
        sauce = "Sauce Tomate Prune";  
  
        toppings.add("Fromage Mozzarella râpé");  
    }  
  
    void couper() {  
        System.out.println("Découper la pizza en tranches carrées");  
    }  
}
```

La pizza Chicago utilise des tomates prunes comme sauce avec une croûte extra-épaisse.



Le style Chicago profond la pizza au plat contient beaucoup de fromage mozzarella !



La pizza de style Chicago remplace également la méthode cut() afin que les morceaux soient coupés en carrés.

Vous avez assez attendu. Il est temps de manger des pizzas !

```
classe publique PizzaTestDrive {
```

```
    public static void main(String[] args) {
        PizzaStore nyStore = new NYPizzaStore(); PizzaStore
        chicagoStore = new ChicagoPizzaStore();

        Pizza pizza = nyStore.orderPizza("fromage"); System.out.println("Ethan a
        commandé une " + pizza.getName() + "\n");

        pizza = chicagoStore.orderPizza("fromage"); System.out.println("Joël a
        commandé une " + pizza.getName() + "\n");
    }
}
```

Nous créons d'abord deux magasins différents.

Nous utilisons un magasin pour préparer la commande d'Ethan...

... et l'autre pour Joel.

Fenêtre d'édition de fichier Aide Vous voulez MootzOnThatPizza ?

```
%java PizzaTestDrive
```

Préparation d'une pizza à la sauce et au fromage de style new-yorkais

Mélanger la pâte...

Ajout de sauce...

Ajout de garnitures :

Fromage Reggiano râpé

Cuire au four pendant 25 minutes à 350 °C. Couper la pizza en tranches diagonales. Placer la pizza dans la boîte officielle PizzaStore.

Ethan a commandé une pizza à la sauce et au fromage de style new-yorkais.

Préparation d'une pizza au fromage à croûte épaisse de style

Chicago Mélanger la pâte...

Ajout de sauce...

Ajout de garnitures :

Fromage mozzarella râpé Cuire au four pendant 25 minutes à 350 °C Couper la pizza en tranches carrées Placer la pizza dans la boîte officielle PizzaStore

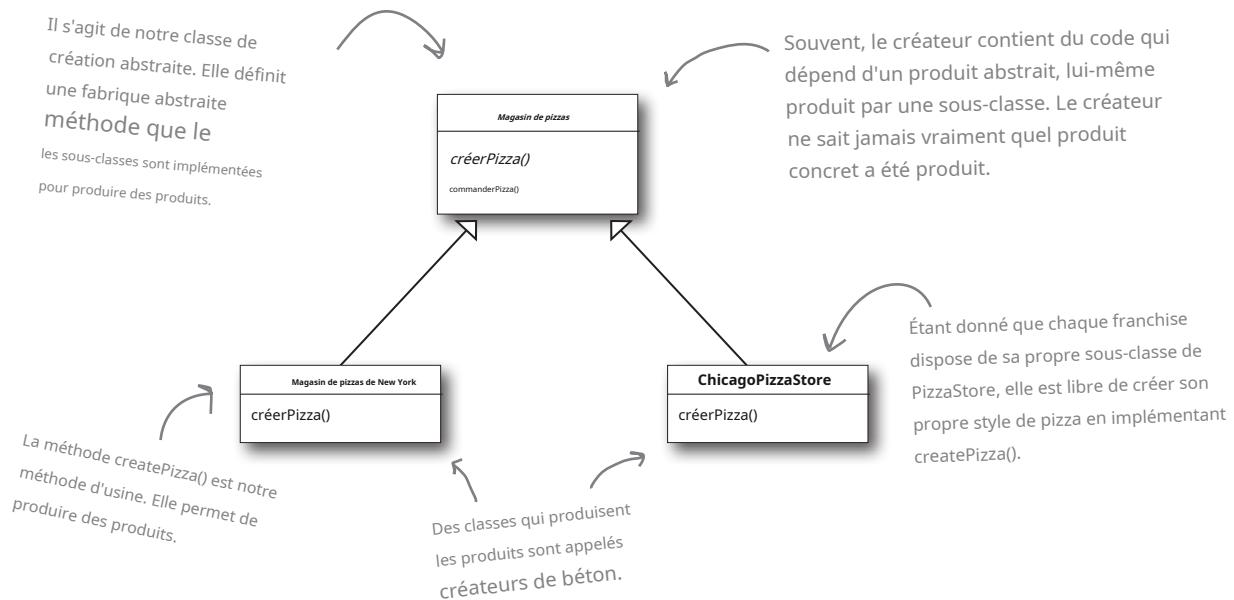
Joel a commandé une pizza au fromage à pâte épaisse de style Chicago

Les deux pizzas sont préparées, les garnitures sont ajoutées, puis les pizzas sont cuites, coupées et mises en boîte. Notre super-classe n'a jamais eu à connaître le détails ; la sous-classe a géré tout cela simplement en instantiant la bonne pizza.

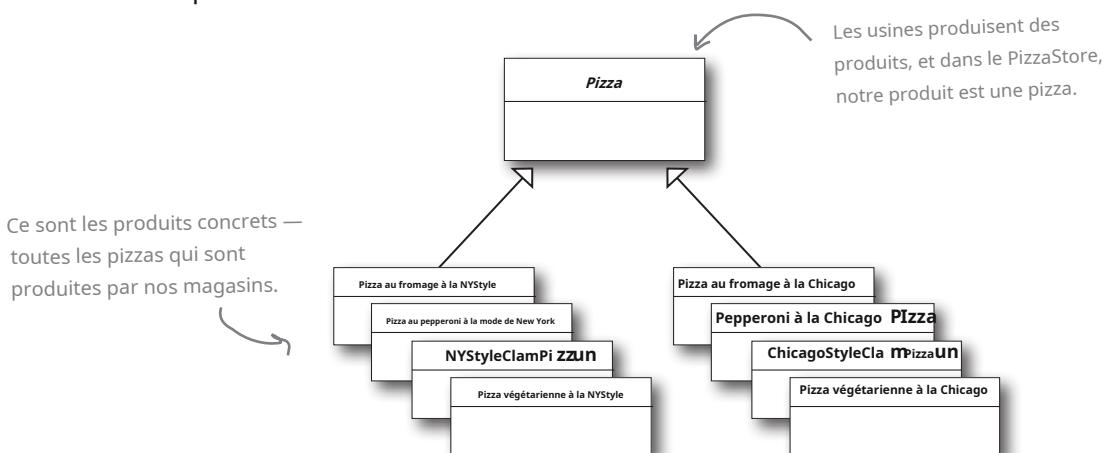
Il est enfin temps de découvrir le modèle de méthode d'usine

Tous les modèles de fabrique encapsulent la création d'objets. Le modèle de méthode de fabrique encapsule la création d'objets en laissant les sous-classes décider quels objets créer. Regardons ces diagrammes de classes pour voir qui sont les acteurs de ce modèle :

La classe Créateur



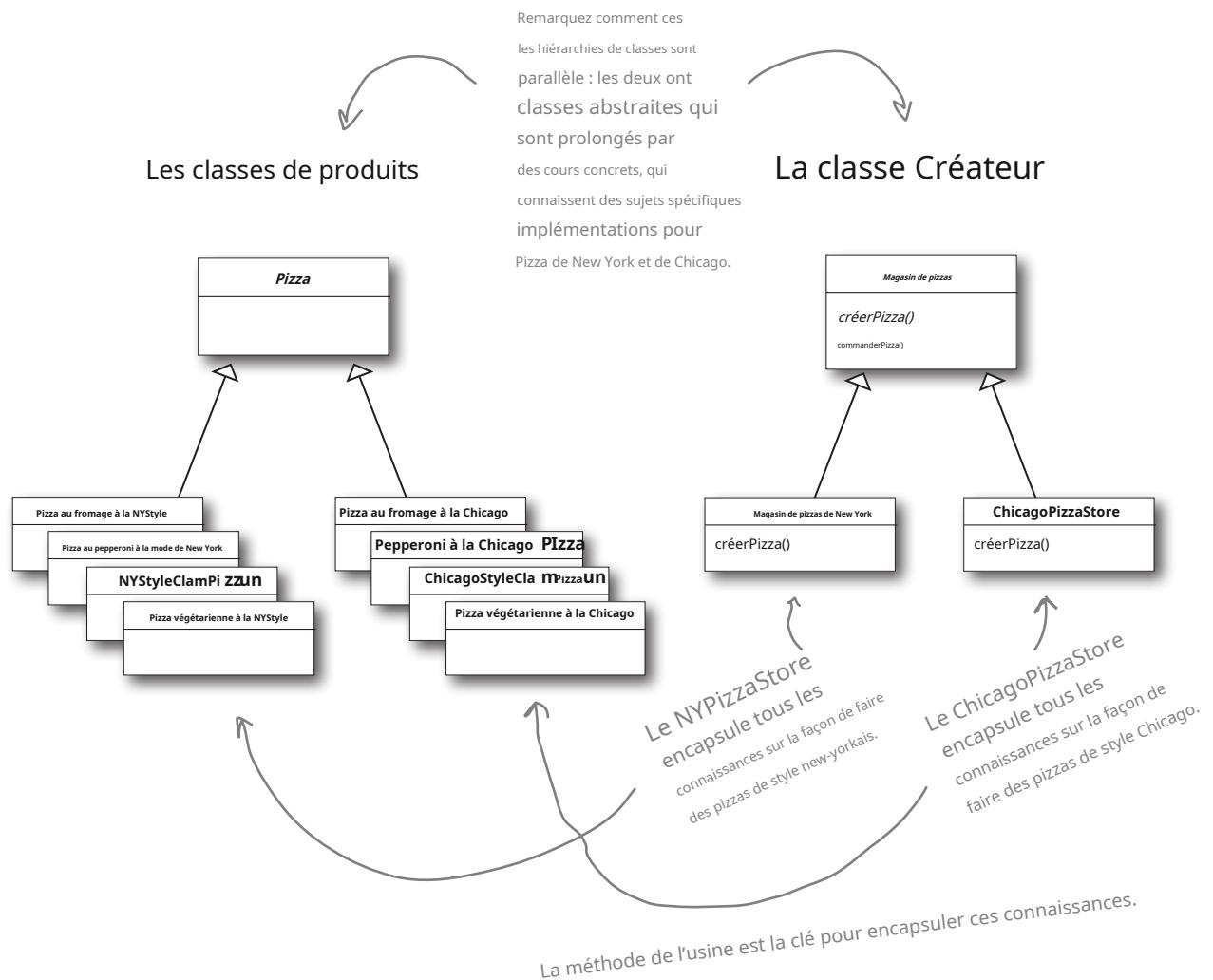
Les classes de produits



Afficher les créateurs et les produits en parallèle

Pour chaque créateur concret, il existe généralement un ensemble complet de produits qu'il crée. Les créateurs de pizzas de Chicago créent différents types de pizzas de style Chicago, les créateurs de pizzas de New York créent différents types de pizzas de style New York, et ainsi de suite. En fait, nous pouvons considérer nos ensembles de classes Creator et leurs classes Product correspondantes comme *hiérarchies parallèles*.

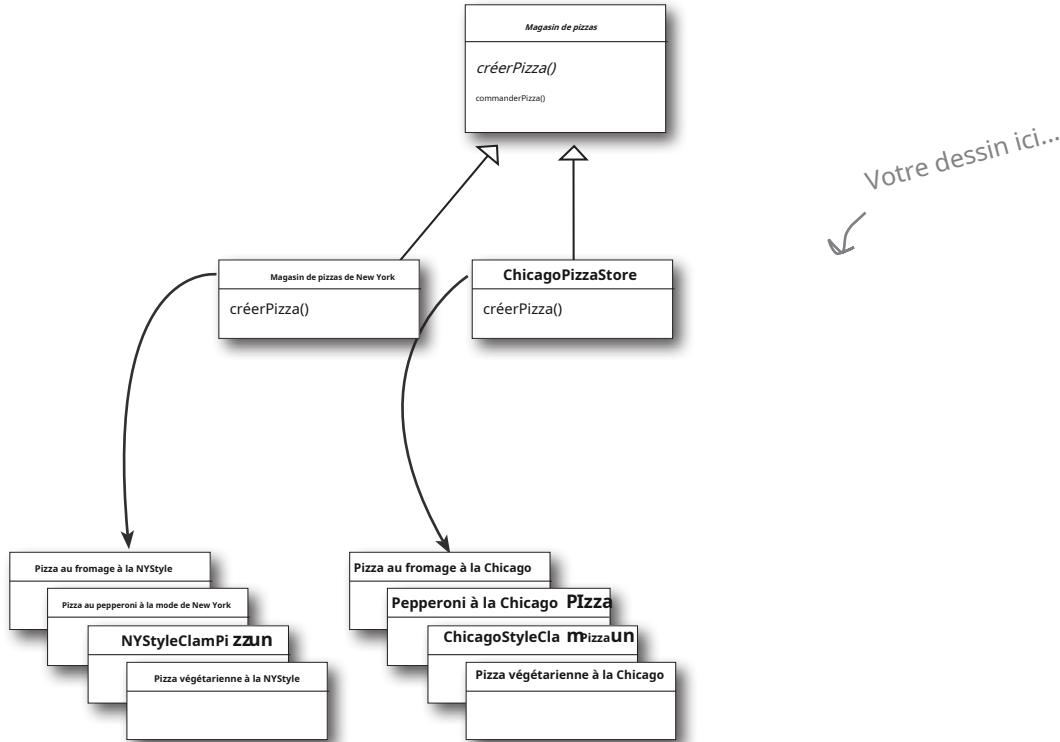
Regardons les deux hiérarchies de classes parallèles et voyons comment elles sont liées :





Puzzle de conception

Nous avons besoin d'un autre type de pizza pour ces Californiens fous (fous dans un *bien* sûr). Dessinez un autre ensemble parallèle de classes dont vous auriez besoin pour ajouter une nouvelle région de Californie à notre PizzaStore.



Ok, maintenant écris les cinq/e plus bizarres choses auxquelles vous pouvez penser pour mettre sur une pizza. Ensuite, vous serez prêt à vous lancer dans la fabrication de pizzas en Californie !

Modèle de méthode d'usine défini

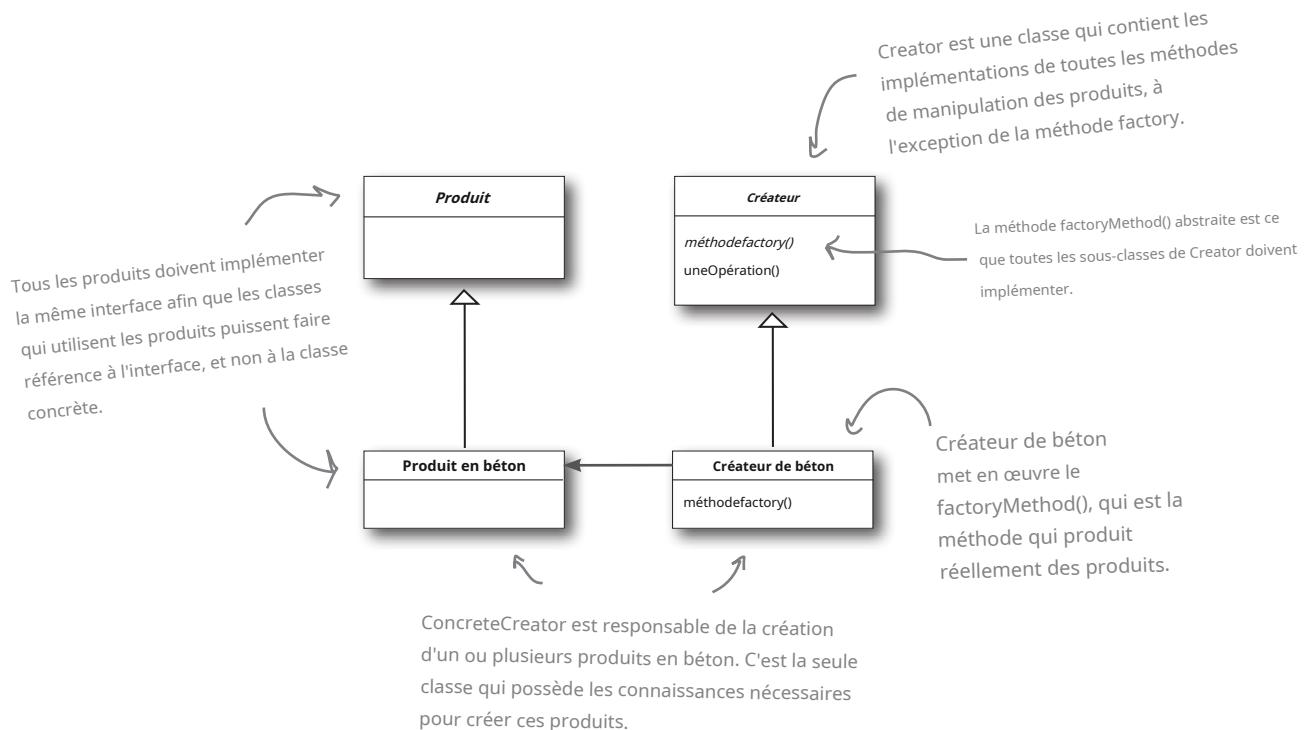
Il est temps de dévoiler la définition officielle du modèle Factory Method :

Le modèle de méthode d'usinedéfinit une interface pour créer un objet, mais permet aux sous-classes de décider quelle classe instancier. La méthode Factory permet à une classe de différer l'instanciation aux sous-classes.

Comme pour toute fabrique, le modèle de méthode de fabrique nous donne un moyen d'encapsuler les instantiations de types concrets. En regardant le diagramme de classes ci-dessous, vous pouvez voir que la classe abstraite Creator vous donne une interface avec une méthode de création d'objets, également connue sous le nom de « méthode de fabrique ». Toutes les autres méthodes implémentées dans la classe abstraite Creator sont écrites pour fonctionner sur les produits produits par la méthode de fabrique. Seules les sous-classes implémentent réellement la méthode de fabrique et créent des produits.

Comme dans la définition officielle, vous entendrez souvent les développeurs dire : « le modèle Factory Method permet aux sous-classes de décider quelle classe instancier. » Étant donné que la classe Creator est écrite sans connaissance des produits réels qui seront créés, nous disons « décider » non pas parce que le modèle permet aux sous-classes *eux-mêmes* de décider, mais plutôt parce que la décision revient en fait à *quelle sous-classe* est utilisée pour créer le produit.

Vous pourriez leur demander ce que signifie « décider », mais nous savons que vous comprenez désormais cela mieux qu'eux !



there are no
Dumb Questions

Q: Quel est l'avantage du modèle de méthode d'usine lorsque vous n'avez qu'un seul ConcreteCreator ?

UN: Le modèle de méthode Factory est utile si vous n'avez qu'un seul créateur concret, car vous découpez l'implémentation du produit de son utilisation. Si vous ajoutez des produits supplémentaires ou modifiez l'implémentation d'un produit, cela n'affectera pas votre créateur (car le créateur n'est pas étroitement couplé à un produit concret).

Q: Est-il exact de dire que nos magasins de New York et de Chicago sont mis en œuvre à l'aide de Simple Factory ? Ils y ressemblent comme deux gouttes d'eau.

UN: Ils sont similaires, mais utilisés de manière différente. Même si l'implémentation de chaque magasin concret ressemble beaucoup à SimplePizzaFactory, n'oubliez pas que les magasins concrets étendent une classe qui a défini createPizza() comme une méthode abstraite. Il appartient à chaque magasin de définir le comportement de la méthode createPizza(). Dans Simple Factory, la fabrique est un autre objet composé avec PizzaStore.

Q: La méthode factory et la classe Creator sont-elles toujours abstraites ?

UN: Non, vous pouvez définir une méthode de fabrique par défaut pour produire un produit concret. Vous disposez alors toujours d'un moyen de créer des produits même s'il n'existe pas de sous-classes de la classe Creator.

Q: Chaque magasin peut fabriquer quatre types de pizzas différents en fonction du type transmis. Est-ce que tous les créateurs de béton fabriquent plusieurs produits ou parfois n'en fabriquent qu'un seul ?

UN: Nous avons implémenté ce que l'on appelle la méthode de fabrique paramétrée. Elle peut créer plusieurs objets en fonction d'un paramètre passé, comme vous l'avez remarqué. Cependant, il arrive souvent qu'une fabrique ne produise qu'un seul objet et ne soit pas paramétrée. Les deux sont des formes valides du modèle.

Q: Vos types paramétrés ne semblent pas « typesafe ». Je passe juste une chaîne ! Et si je demandais une « CalmPizza » ?

UN: Vous avez certainement raison, et cela provoquerait ce que nous appelons dans le métier une « erreur d'exécution ». Il existe plusieurs autres techniques plus sophistiquées qui peuvent être utilisées pour rendre les paramètres plus « sûrs en termes de type », c'est-à-dire pour garantir que les erreurs dans les paramètres peuvent être détectées au moment de la compilation. Par exemple, vous pouvez créer des objets qui représentent les types de paramètres, utiliser des constantes statiques ou utiliser des énumérations.

Q: Je suis encore un peu confus quant à la différence entre Simple Factory et Factory Method. Elles se ressemblent beaucoup, sauf que dans Factory Method, la classe qui renvoie la pizza est une sous-classe. Pouvez-vous expliquer ?

UN: Vous avez raison de dire que les sous-classes ressemblent beaucoup à Simple Factory. Cependant, considérez Simple Factory comme une solution unique, tandis qu'avec Factory Method, vous créez un cadre qui permet aux sous-classes de décider quelle implémentation sera utilisée. Par exemple, la méthode orderPizza() dans le modèle Factory Method fournit un cadre général pour la création de pizzas qui s'appuie sur une méthode factory pour créer réellement les classes concrètes qui entrent dans la fabrication d'une pizza. En sous-classant la classe PizzaStore, vous décidez quels produits concrets entrent dans la fabrication de la pizza que orderPizza() renvoie. Comparez cela avec Simple Factory, qui vous donne un moyen d'encapsuler la création d'objets, mais ne vous donne pas la flexibilité de Factory Method car il n'y a aucun moyen de faire varier les produits que vous créez.



Gourou et étudiant...

Gourou:Parlez-moi de votre formation.

Étudiant:Gourou, j'ai poussé plus loin mon étude sur « encapsuler ce qui varie ».

Gourou:Continue...

Étudiant:*j'ai appris qu'il est possible d'encapsuler le code qui crée des objets. Lorsque vous avez du code qui instancie des classes concrètes, c'est un domaine qui évolue fréquemment. J'ai appris une technique appelée « factories » qui permet d'encapsuler ce comportement d'instanciation.*

Gourou:*Et ces « usines », à quoi servent-elles ?*

Étudiant:*Il y en a beaucoup. En plaçant tout mon code de création dans un seul objet ou une seule méthode, j'évite les doublons dans mon code et je ne fournis qu'un seul endroit pour effectuer la maintenance. Cela signifie également que les clients ne dépendent que des interfaces plutôt que des classes concrètes requises pour instancier les objets. Comme je l'ai appris au cours de mes études, cela me permet de programmer sur une interface, et non sur une implémentation, ce qui rend mon code plus flexible et extensible à l'avenir.*

Gourou:*Oui, vos instincts OO se développent. Avez-vous des questions à poser à votre gourou aujourd'hui ?*

Étudiant:*Guru, je sais qu'en encapsulant la création d'objets, je code des abstractions et je découpe mon code client des implementations réelles. Mais mon code d'usine doit toujours utiliser des classes concrètes pour instancier des objets réels. Est-ce que je ne me fais pas avoir ?*

Gourou:*La création d'objets est une réalité de la vie. Nous devons créer des objets, sinon nous ne créerions jamais une seule application Java. Mais, en connaissant cette réalité, nous pouvons concevoir notre code de manière à ce que nous ayons rassemblé ce code de création comme le mouton dont vous tireriez la laine sur vos yeux. Une fois rassemblé, nous pouvons protéger et prendre soin du code de création. Si nous laissons notre code de création se déchaîner, nous ne récolterons jamais sa « laine ».*

Étudiant:*Gourou, je vois la vérité là-dedans.*

Gourou:*Comme je m'y attendais. Maintenant, s'il vous plaît, allez méditer sur les dépendances des objets.*

Sharpen your pencil



Imaginons que vous n'avez jamais entendu parler d'une fabrique OO. Voici une version « très dépendante » de PizzaStore qui n'utilise pas de fabrique. Nous avons besoin que vous comptiez le nombre de classes de pizza concrètes dont dépend cette classe. Si vous ajoutiez des pizzas de style californien à ce PizzaStore, de combien de classes dépendrait-il alors ?

```

classe publique DependentPizzaStore {

    public Pizza createPizza(Style de chaîne, Type de chaîne) {
        Pizza pizza = null; si
        (style.equals("NY")) {
            si (type.equals("fromage")) {
                pizza = nouvelle NYStyleCheesePizza(); }
            else if (type.equals("végétarien")) {
                pizza = nouvelle NYStyleVeggiePizza(); }
            sinon si (type.equals("clam")) {
                pizza = nouvelle NYStyleClamPizza(); }
            sinon si (type.equals("pepperoni")) {
                pizza = nouveau NYStylePepperoniPizza(); }
            }
        } sinon si (style.equals("Chicago")) {
            si (type.equals("fromage")) {
                pizza = nouvelle ChicagoStyleCheesePizza(); }
            else if (type.equals("végétarien")) {
                pizza = nouvelle ChicagoStyleVeggiePizza(); }
            else if (type.equals("clam")) {
                pizza = nouvelle ChicagoStyleClamPizza(); }
            sinon si (type.equals("pepperoni")) {
                pizza = nouvelle ChicagoStylePepperoniPizza(); }
            }
        } autre {
            System.out.println("Erreur : type de pizza non valide "); return
            null;
        }
        pizza.préparer();
        pizza.cuire();
        pizza.couper();
        boîte à pizza();
        retourner la pizza;
    }
}

```

Gère tous les Pizzas à la new-yorkaise

Gère tous les Pizzas à la Chicago

Vous pouvez écrire vos réponses ici :

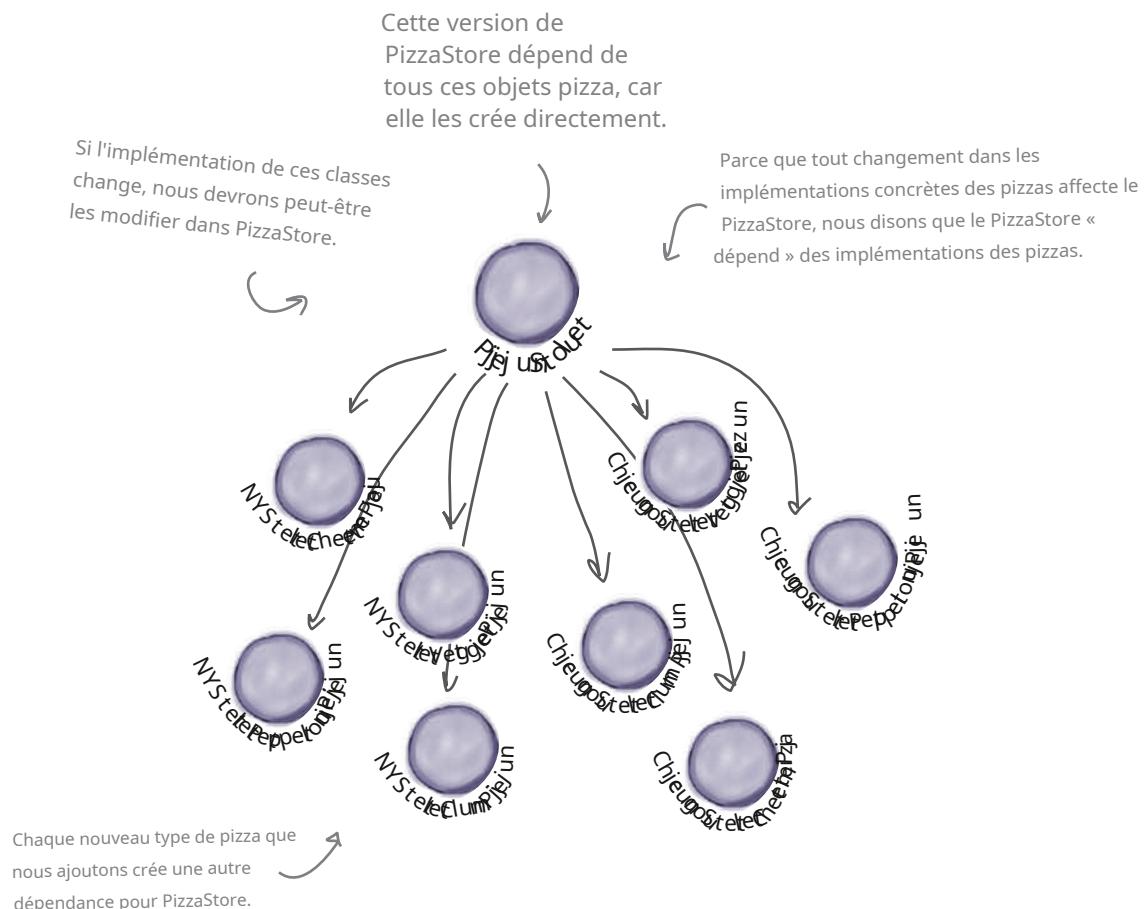
nombre _____

numéro avec La Californie aussi _____

Regard sur les dépendances des objets

Lorsque vous instanciez directement un objet, vous dépendez de sa classe concrète. Jetez un œil à notre PizzaStore très dépendant une page en arrière. Il crée tous les objets pizza directement dans la classe PizzaStore au lieu de déléguer à une usine.

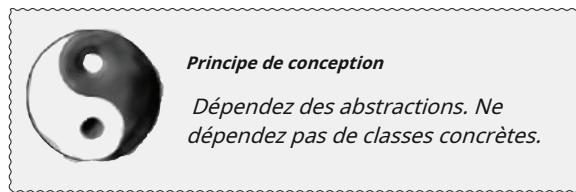
Si nous dessinons un diagramme représentant cette version du PizzaStore et tous les objets dont il dépend, voici à quoi cela ressemble :



Le principe d'inversion de dépendance

Il devrait être assez clair que la réduction des dépendances aux classes concrètes dans notre code est une « bonne chose ». En fait, nous avons un principe de conception OO qui formalise cette notion ; il a même un grand nom formel : *Principe d'inversion de dépendance*.

Voici le principe général :



Encore une phrase que vous pouvez utiliser pour impressionner les dirigeants présents dans la salle ! Votre augmentation compensera largement le coût de ce livre et vous gagnerez l'admiration de vos collègues développeurs.

Au premier abord, ce principe ressemble beaucoup à « Programmer vers une interface, pas vers une implémentation », n'est-ce pas ? C'est similaire, mais le principe d'inversion de dépendance fait une déclaration encore plus forte sur l'abstraction. Il suggère que nos composants de haut niveau ne devraient pas dépendre de nos composants de bas niveau ; ils devraient plutôt *être dépendant d'abstractions*.

Mais qu'est-ce que ça veut dire ?

Commençons par examiner à nouveau le diagramme de la pizzeria de la page précédente. PizzaStore est notre « composant de haut niveau » et les implémentations de pizza sont nos « composants de bas niveau », et il est clair que PizzaStore dépend des classes de pizza concrètes.

Ce principe nous dit que nous devrions plutôt écrire notre code de manière à ce que nous dépendions d'abstractions et non de classes concrètes. Cela vaut aussi bien pour nos modules de haut niveau que pour nos modules de bas niveau.

Mais comment faire ? Réfléchissons à la façon dont nous appliquerions ce principe à notre implémentation très dépendante de PizzaStore...

Un composant de « haut niveau » est une classe dont le comportement est défini en termes d'autres composants de « bas niveau ».

Par exemple, PizzaStore est un composant de haut niveau car son comportement est défini en termes de pizzas : il crée tous les différents objets pizza, les prépare, les cuit, les coupe et les met en boîte, tandis que les pizzas qu'il utilise sont des composants de bas niveau.

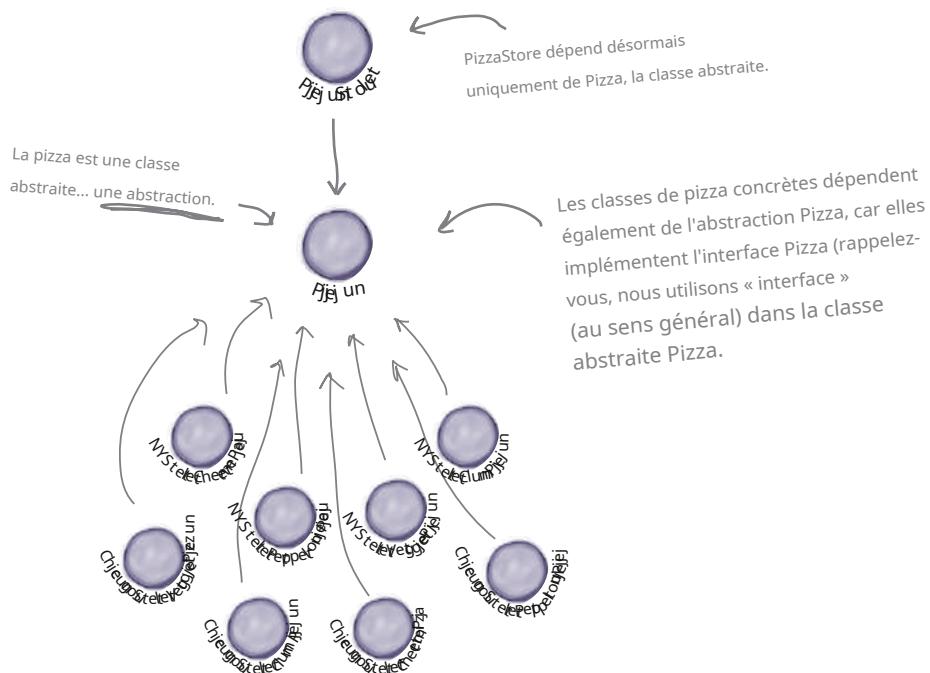
Application du principe

Maintenant, le problème principal avec le PizzaStore très dépendant est qu'il dépend de chaque type de pizza car il instancie en fait des types concrets dans sa méthode `orderPizza()`.

Bien que nous ayons créé une abstraction, `Pizza`, nous créons néanmoins des pizzas concrètes dans ce code, nous ne tirons donc pas beaucoup de profit de cette abstraction.

Comment pouvons-nous extraire ces instantiations de la méthode `orderPizza()`? Comme nous le savons, le modèle de méthode d'usine nous permet de faire exactement cela.

Ainsi, après avoir appliqué le modèle de méthode d'usine, notre diagramme ressemble à ceci :



Après avoir appliqué la méthode Factory, vous remarquerez que notre composant de haut niveau, le `PizzaStore`, et nos composants de bas niveau, les pizzas, dépendent tous deux de `Pizza`, l'abstraction. La méthode Factory n'est pas la seule technique permettant d'adhérer au principe d'inversion de dépendance, mais c'est l'une des plus puissantes.



Où est l'« inversion » dans le principe d'inversion de dépendance ?

Le terme « inversion » dans le nom Principe d'inversion de dépendances est utilisé parce qu'il inverse la façon dont vous envisagez généralement votre conception OO. Regardez le diagramme de la page précédente. Notez que les composants de bas niveau dépendent désormais d'une abstraction de niveau supérieur. De même, le composant de haut niveau est également lié à la même abstraction. Ainsi, le graphique de dépendances de haut en bas que nous avons dessiné quelques pages plus haut s'est inversé, les modules de haut et de bas niveau dépendant désormais de l'abstraction.

Examinons également la réflexion qui sous-tend le processus de conception typique et voyons comment l'introduction du principe peut inverser notre façon de penser la conception...

Inverser votre façon de penser...



Bon, vous devez donc ouvrir une pizzeria. Quelle est la première idée qui vous vient à l'esprit ?

Bon, vous commencez par le haut et vous suivez les choses jusqu'aux classes concrètes. Mais, comme vous l'avez vu, vous ne voulez pas que votre pizzeria connaisse les types de pizzas concrètes, car elle dépendra alors de toutes ces classes concrètes !

Maintenant, « inversons » votre réflexion... au lieu de commencer par le haut, commencez par les pizzas et réfléchissez à ce que vous pouvez abstraire.

C'est vrai ! Vous pensez à l'abstraction *Pizza*. Alors maintenant, revenez en arrière et réfléchissez à nouveau à la conception de la pizzeria.

C'est presque fini. Mais pour cela, vous devrez vous appuyer sur une usine pour extraire ces classes concrètes de votre pizzeria. Une fois que vous avez fait cela, vos différents types de pizzas concrètes ne dépendront que d'une abstraction, tout comme votre pizzeria. Nous avons pris une conception où la pizzeria dépendait de classes concrètes et avons inversé ces dépendances (ainsi que votre réflexion).

Quelques lignes directrices pour vous aider à suivre le Principe...

Les directives suivantes peuvent vous aider à éviter les conceptions OO qui violent le principe d'inversion de dépendance :

fAucune variable ne doit contenir une référence à une classe concrète.

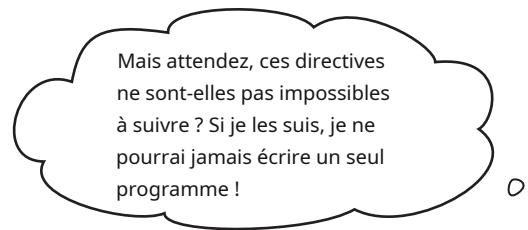
Si vous utilisez `nouveau`, vous aurez une référence à une classe concrète. Utilisez une fabrique pour contourner cela !

fAucune classe ne doit dériver d'une classe concrète.

Si vous dérivez d'une classe concrète, vous dépendez d'une classe concrète. Dérivez d'une abstraction, comme une interface ou une classe abstraite.

f Aucune méthode ne doit remplacer une méthode implémentée d'une de ses classes de base.

Si vous remplacez une méthode implémentée, votre classe de base n'était pas vraiment une abstraction au départ. Les méthodes implémentées dans la classe de base sont censées être partagées par toutes vos sous-classes.



Vous avez tout à fait raison ! Comme beaucoup de nos principes, il s'agit d'une ligne directrice à suivre plutôt que d'une règle à suivre en permanence. Il est clair que chaque programme Java jamais écrit viole ces directives !

Mais si vous assimilez ces règles et que vous les gardez à l'esprit lorsque vous concevez, vous saurez quand vous violez le principe et vous aurez une bonne raison de le faire. Par exemple, si vous avez une classe qui n'est pas susceptible de changer, et que vous le savez, ce n'est pas la fin du monde si vous instanciez une classe concrète dans votre code. Pensez-y : nous instancions des objets String tout le temps sans réfléchir. Est-ce que cela viole le principe ? Oui. Est-ce correct ? Oui. Pourquoi ? Parce que String est très peu susceptible de changer.

Si, en revanche, une classe que vous écrivez est susceptible de changer, vous disposez de bonnes techniques comme la méthode Factory pour encapsuler ce changement.



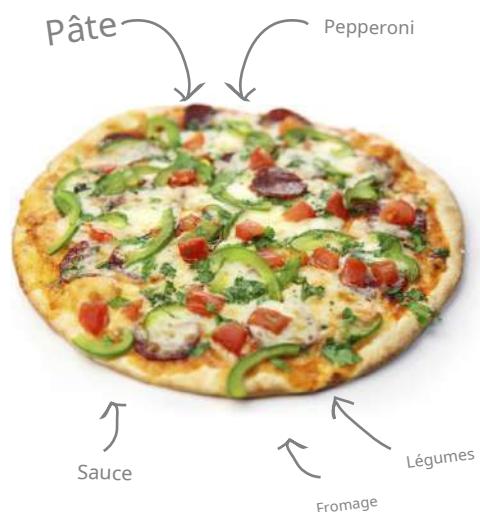
familles des ingrédients

Pendant ce temps, de retour à la pizzeria...

La conception de la pizzeria prend vraiment forme : elle dispose d'un cadre flexible et adhère bien aux principes de conception.

Maintenant, la clé du succès d'Objectville Pizza a toujours été des ingrédients frais et de qualité, et ce que vous avez découvert, c'est qu'avec le nouveau cadre, vos franchises ont suivi votre *procédures*, mais quelques franchises ont remplacé les ingrédients de qualité inférieure dans leurs pizzas pour réduire les coûts et augmenter leurs marges. Vous savez que vous devez faire quelque chose, car à long terme, cela va nuire à la marque Objectville !

C'est-à-dire la cuisson,
la découpe, la
la boîte, etc.



Assurer la cohérence de vos ingrédients

Alors, comment allez-vous vous assurer que chaque franchise utilise des ingrédients de qualité ? Vous allez construire une usine qui les produit et les expédie à vos franchises !

Il n'y a qu'un seul problème avec ce plan : les franchises sont situées dans des régions différentes et ce qui est de la sauce rouge à New York ne l'est pas à Chicago. Vous avez donc un ensemble d'ingrédients qui doivent être expédiés à New York et un autre qui doit être livré à Chicago. *différent ensemble* qui doit être expédié à Chicago. Regardons de plus près :



Chicago

Menu Pizza

- Pizza au fromage**
Sauce Tomate Prune, Mozzarella, Parmesan, Origan
- Pizza végétarienne**
Sauce tomate prune, mozzarella, parmesan, aubergine, épinards, olives noires
- Pizza aux palourdes**
Sauce tomate prune, mozzarella, parmesan, palourdes
- Pizza au pepperoni**
Sauce tomate prune, mozzarella, parmesan, aubergine, épinards, olives noires, pepperoni

Nous avons le même produit familles (pâte, sauce, fromage, légumes, viandes) mais différent implémentations basé sur la région.



New York

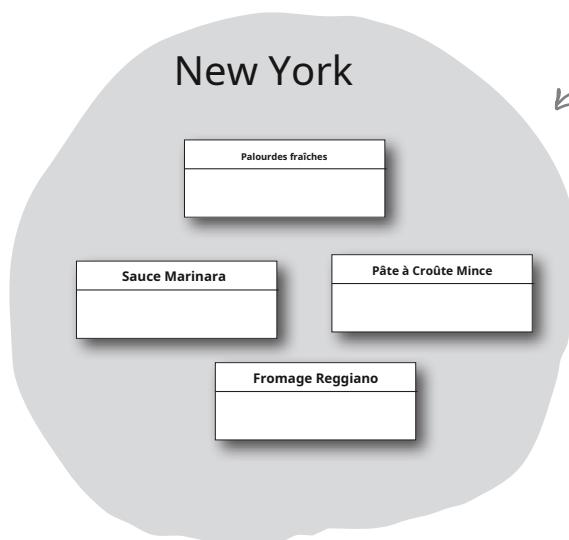
Menu Pizza

- Pizza au fromage**
Sauce Marinara, Reggiano, Ail
- Pizza végétarienne**
Sauce Marinara, Reggiano, Champignons, Oignons, Poivrons Rouges
- Pizza aux palourdes**
Sauce Marinara, Reggiano, Palourdes Fraîches
- Pizza au pepperoni**
Sauce Marinara, Reggiano, Champignons, Oignons, Poivrons Rouges, Pepperoni

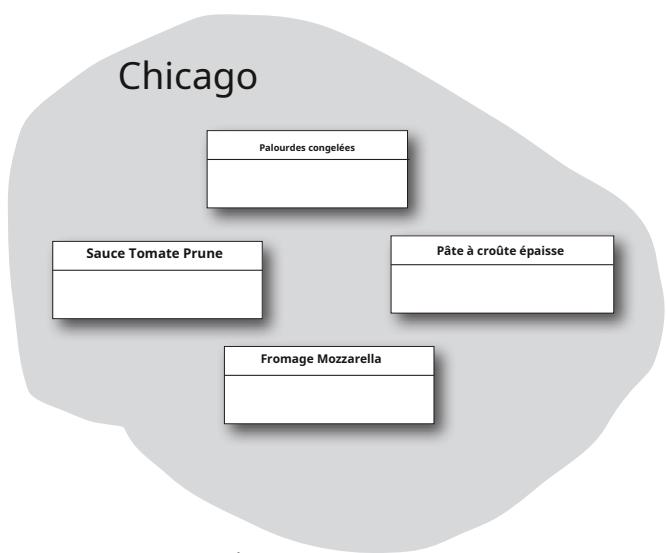
Familles d'ingrédients...

New York utilise un ensemble d'ingrédients et Chicago un autre. Étant donné la popularité d'Objectville Pizza, il ne faudra pas longtemps avant que vous ayez également besoin d'expédier un autre ensemble d'ingrédients régionaux en Californie, et quelle est la prochaine étape ? Austin ?

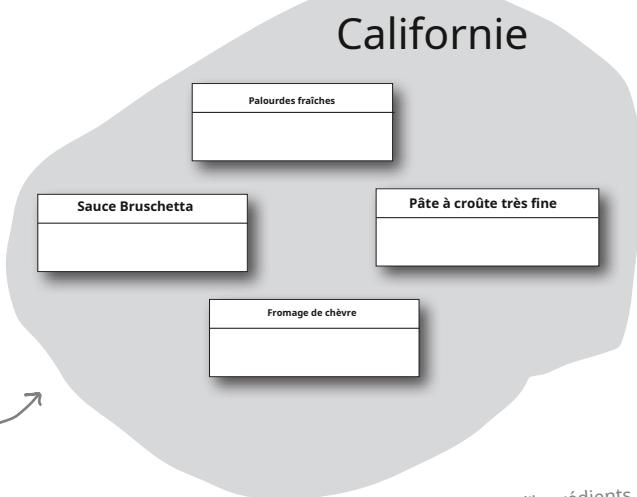
Pour que cela fonctionne, vous allez devoir comprendre comment gérer les familles d'ingrédients.



Chaque famille se compose d'un type de pâte, d'un type de sauce, d'un type de fromage et d'une garniture de fruits de mer (ainsi que de quelques autres que nous n'avons pas montrés, comme des légumes et des épices).



Toutes les pizzas d'Objectville sont fabriquées à partir des mêmes composants, mais chaque région a une implémentation différente de ces composants.



Au total, ces trois régions constituent des familles d'ingrédients, chaque région mettant en œuvre une famille complète d'ingrédients.

Construire les usines d'ingrédients

Nous allons maintenant construire une usine pour créer nos ingrédients ; l'usine sera chargée de créer chaque ingrédient de la famille d'ingrédients. En d'autres termes, l'usine devra créer de la pâte, de la sauce, du fromage, etc. Vous verrez bientôt comment nous allons gérer les différences régionales.

Commençons par définir une interface pour l'usine qui va créer tous nos ingrédients :

```
interface publique PizzaIngredientFactory {
```

```
    public Pâte createDough(); public Sauce  
    createSauce(); public Fromage  
    createCheese(); public Légumes[]  
    createVeggies(); public Pepperoni  
    createPepperoni(); public Palourdes  
    createClam();
```

```
}
```

```
)
```

Beaucoup de nouvelles classes
ici, une par ingrédient.



Pour chaque ingrédient, nous définissons une
méthode de création dans notre interface.

Avec cette interface, voici ce que nous allons faire :

- 1 Créez une usine pour chaque région. Pour ce faire, vous allez créer une sous-classe de PizzaIngredientFactory qui implémente chaque méthode de création.
- 2 Implémentez un ensemble de classes d'ingrédients à utiliser avec l'usine, comme ReggianoCheese, RedPeppers et ThickCrustDough. Ces classes peuvent être partagées entre les régions, le cas échéant.
- 3 Ensuite, nous devons encore connecter tout cela en intégrant nos nouvelles usines d'ingrédients dans notre ancien code PizzaStore.

Construction de l'usine d'ingrédients de New York

Ok, voici l'implémentation pour l'usine d'ingrédients de New York. Cette usine est spécialisée dans la sauce Marinara, le fromage Reggiano, les palourdes fraîches, etc.

L'usine d'ingrédients NY implémente l'interface pour toutes les usines d'ingrédients.

la classe publique NYPizzaIngredientFactory implémente PizzaIngredientFactory {

```
public Pâte createDough() {
    renvoie un nouveau ThinCrustDough();
}
```

Pour chaque ingrédient de la famille d'ingrédients, nous créons la version new-yorkaise.

```
Sauce publique createSauce() {
    renvoie la nouvelle MarinaraSauce();
}
```

```
public Fromage createCheese() {
    renvoie un nouveau ReggianoCheese();
}
```

```
public Veggies[] createVeggies() {
    Légumes veggies[] = { new Garlic(), new Onion(), new Mushroom(), new RedPepper() }; return veggies;
}
```

Pour les légumes, nous renvoyons un tableau de légumes. Ici, nous avons codé en dur les légumes. Nous pourrions rendre cela plus sophistiqué, mais cela n'ajoute rien à l'apprentissage du modèle d'usine, nous allons donc le garder simple.

```
public Pepperoni createPepperoni() {
    renvoie un nouveau SlicedPepperoni();
}
```



Le meilleur pepperoni tranché. Ce plat est partagé entre New York et Chicago. Assurez-vous de l'utiliser sur la page suivante lorsque vous mettrez en œuvre vous-même l'usine de Chicago.

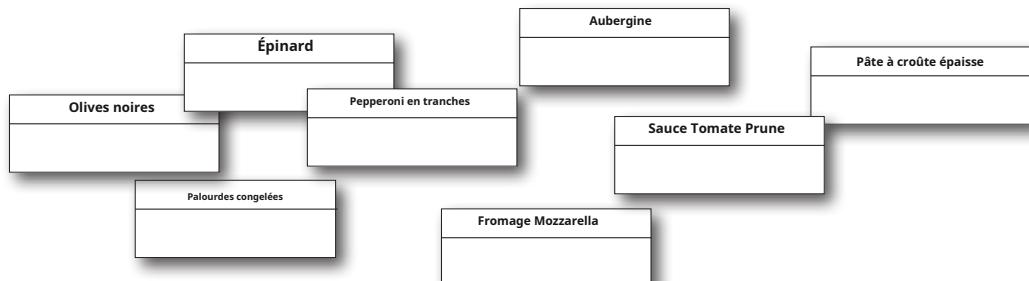
```
palourdes publiques createClam() {
    renvoie un nouveau FreshClams();
}
```

New York est située sur la côte, on y trouve des palourdes fraîches. Chicago doit se contenter de palourdes congelées.



Sharpen your pencil

Écrivez la ChicagoPizzaIngredientFactory. Vous pouvez référencer les classes ci-dessous dans votre implémentation :



Retravailler les pizzas...

Nos usines sont toutes en marche et prêtes à produire des ingrédients de qualité ; il ne nous reste plus qu'à retravailler nos pizzas pour qu'elles n'utilisent que des ingrédients produits en usine. Nous commencerons par notre cours de pizza abstrait :

```
classe abstraite publique Pizza {
```

```
    Nom de la chaîne ;
```

```
    Pâte à pâte;
```

```
    Sauce sauce;
```

```
    Légumes légumes[];
```

```
    Fromage fromage;
```

```
    Pepperoni pepperoni;
```

```
    Palourdes;
```

```
    résumé void prepare();
```

Chaque pizza contient un ensemble d'ingrédients qui sont utilisés dans sa préparation.

```
void cuire() {
```

```
    System.out.println("Cuire au four pendant 25 minutes à 350");
```

```
}
```

```
void couper() {
```

```
    System.out.println("Découper la pizza en tranches diagonales");
```

```
}
```

```
boîte vide() {
```

```
    System.out.println("Placez la pizza dans la boîte officielle PizzaStore");
```

```
}
```

```
void setName(Chaîne nom) {
```

```
    ceci.nom = nom;
```

```
}
```

Nous avons maintenant rendu la méthode de préparation abstraite. C'est ici que nous allons collecter les ingrédients nécessaires à la pizza, qui viendront bien sûr de l'usine d'ingrédients.

```
Chaîne getName() {
```

```
    renvoyer le nom;
```

```
}
```

```
chaîne publique toString() {
```

```
    // code pour imprimer la pizza ici
```

```
}
```

```
}
```

Nos autres méthodes restent les mêmes, à l'exception de la méthode de préparation.

Retravailler les pizzas, suite...

Maintenant que vous avez une classe de pizza abstraite sur laquelle travailler, il est temps de créer les pizzas de style New York et Chicago, mais cette fois-ci, les ingrédients seront obtenus directement de l'usine. L'époque où les franchisés lésinaient sur les ingrédients est révolue !

Lorsque nous avons écrit le code de la méthode Factory, nous avions une classe NYCheesePizza et une classe ChicagoCheesePizza. Si vous regardez les deux classes, la seule chose qui diffère est l'utilisation d'ingrédients régionaux. Les pizzas sont faites de la même manière (pâte + sauce + fromage). Il en va de même pour les autres pizzas : Veggie, Clam, etc. Elles suivent toutes les mêmes étapes de préparation ; elles contiennent juste des ingrédients différents.

Donc, ce que vous verrez, c'est que nous n'avons vraiment pas besoin de deux classes pour chaque pizza ; l'usine d'ingrédients va gérer les différences régionales pour nous.

Voici la CheesePizza :

```
classe publique CheesePizza étend Pizza {  
    PizzaIngredientFactory ingrédientFactory;  
  
    public CheesePizza(PizzaIngredientFactory ingrédientFactory) {  
        ceci.ingredientFactory = ingrédientFactory;  
    }  
  
    void préparer() {  
        System.out.println("Préparation " + nom); pâte =  
        ingredientFactory.createDough(); sauce =  
        ingredientFactory.createSauce(); fromage =  
        ingredientFactory.createCheese();  
    }  
}
```

Pour faire une pizza maintenant, nous avons besoin d'une usine pour fournir les ingrédients. Ainsi, chaque classe Pizza reçoit une usine transmise à son constructeur, et elle est stockée dans un variable d'instance.



C'est ici que la magie opère !

La méthode prepare() crée une pizza au fromage et, chaque fois qu'elle a besoin d'un ingrédient, elle demande à l'usine de le produire.





Le code de près

Le code Pizza utilise l'usine avec laquelle il a été composé pour produire les ingrédients utilisés dans la pizza. Les ingrédients produits dépendent de l'usine que nous utilisons. La classe Pizza ne s'en soucie pas ; elle sait comment faire des pizzas. Maintenant, elle est découpée des différences entre les ingrédients régionaux et peut être facilement réutilisée lorsqu'il existe des usines pour Austin, Nashville et au-delà.

sauce = ingrédientFactory.createSauce();

Nous mettons en place le
Exemple de pizza
variable pour faire référence à
la sauce spécifique
utilisé dans cette pizza.

Il s'agit de notre usine d'ingrédients.
La classe Pizza ne se soucie pas de
l'usine utilisée, tant qu'il s'agit d'une
usine d'ingrédients.

La méthode createSauce() renvoie la sauce utilisée dans sa
région. S'il s'agit d'une fabrique d'ingrédients de New York,
nous obtenons alors de la sauce marinara.

Jetons également un œil à la ClamPizza :

**classe publique ClamPizza étend Pizza {
 PizzaIngredientFactory ingrédientFactory;**

ClamPizza cache également
une usine d'ingrédients.

```
public ClamPizza(PizzaIngredientFactory ingrédientFactory) {  
    ceci.ingredientFactory = ingrédientFactory;  
}
```

```
void préparer() {  
    System.out.println("Préparation " + nom); pâte =  
    ingredientFactory.createDough(); sauce =  
    ingredientFactory.createSauce(); fromage =  
    ingredientFactory.createCheese(); palourde =  
    ingredientFactory.createClam();
```

```
}
```

Pour faire une pizza aux palourdes, la
méthode prepare() collecte les bonnes
ingrédients de son usine locale.

S'il s'agit d'une usine de New York, les
palourdes seront fraîches ; si c'est celle
de Chicago, elles seront congelées.

Visite de nos pizzerias

Nous y sommes presque ; il nous reste juste à faire un petit tour dans nos magasins franchisés pour nous assurer qu'ils utilisent les bonnes pizzas. Nous devons également leur donner une référence à leurs usines d'ingrédients locales :

classe publique NYPizzaStore étend PizzaStore {

Pizza protégée createPizza(élément de chaîne) {

**Pizza pizza = null; PizzaIngredientFactory
ingredientFactory =
nouvelle NYPizzaIngredientFactory();**

si (élément.equals("fromage")) {

**pizza = new CheesePizza(ingredientFactory);
pizza.setName("Pizza au fromage de style new-yorkais");**

} sinon si (élément.equals("veggie")) {

**pizza = new VeggiePizza(ingredientFactory);
pizza.setName("Pizza végétarienne à la new-yorkaise");**

} sinon si (élément.equals("clam")) {

**pizza = new ClamPizza(ingredientFactory);
pizza.setName("Pizza aux palourdes à la new-yorkaise");**

} sinon si (élément.equals("pepperoni")) {

**pizza = new PepperoniPizza(ingredientFactory);
pizza.setName("Pizza au pepperoni façon new-yorkaise");**

}

retourner la pizza;

}

Le magasin de New York est composé d'une usine d'ingrédients pour pizzas de New York. Celle-ci servira à produire les ingrédients de toutes les pizzas de style new-yorkais.



Nous passons maintenant pour chaque pizza l'usine qui doit servir à produire ses ingrédients.



Revenez une page en arrière et assurez-vous de bien comprendre comment la pizza et l'usine fonctionnent ensemble !



Pour chaque type de Pizza, nous instancions une nouvelle Pizza et lui donnons l'usine dont elle a besoin pour obtenir ses ingrédients.



Comparez cette version de la méthode createPizza() à celle de l'implémentation de la méthode Factory plus haut dans le chapitre.

Qu'avons-nous fait ?

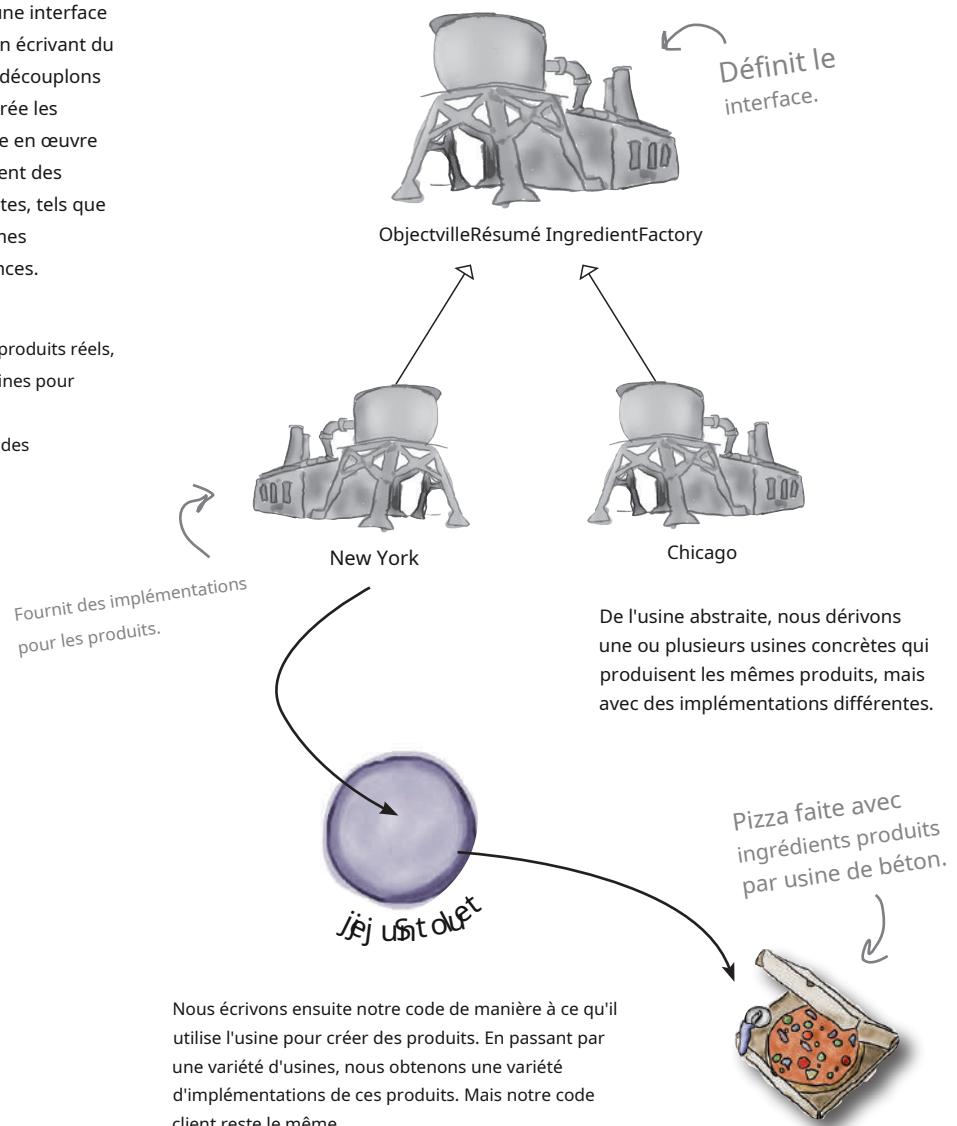
C'était toute une série de changements de code ; qu'avons-nous fait exactement ?

Nous avons fourni un moyen de créer une famille d'ingrédients pour les pizzas en introduisant un nouveau type d'usine appelée *Usine abstraite*.

Une fabrique abstraite nous fournit une interface pour créer une famille de produits. En écrivant du code qui utilise cette interface, nous découplons notre code de la fabrique réelle qui crée les produits. Cela nous permet de mettre en œuvre une variété de fabriques qui produisent des produits destinés à différents contextes, tels que différentes régions, différents systèmes d'exploitation ou différentes apparences.

Parce que notre code est découplé des produits réels, nous pouvons remplacer différentes usines pour obtenir des comportements différents (comme prendre de la marinara au lieu des tomates prunes).

Une fabrique abstraite fournit une interface pour une famille de produits. Qu'est-ce qu'une famille ? Dans notre cas, il s'agit de tout ce dont nous avons besoin pour faire une pizza : pâte, sauce, fromage, viandes et légumes.



Plus de pizza pour Ethan et Joel...

Ethan et Joe ne peuvent pas se lasser de la pizza Objectville ! Ce qu'ils ne savent pas, c'est que maintenant leurs commandes utilisent le nouveau facteur d'ingrédients, c'est-à-dire les s. Alors maintenant, quand ils commandent...

Derrière
les scènes



La première partie du processus de commande n'a pas changé du tout. Suivons à nouveau la commande d'Ethan :

- 1 **On a besoin d'un NYPizzaStore :**

```
Pizzeria nyPizzaStore = new NYPizzaStore();
```

Crée une instance de NYPizzaStore.

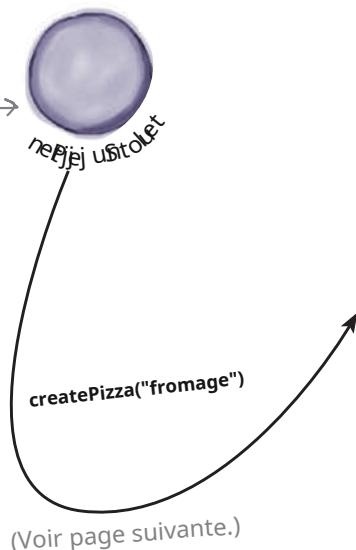
- 2 **Maintenant que nous avons un magasin, nous pouvons prendre une commande :**

```
nyPizzaStore.orderPizza("fromage");
```

La méthode orderPizza() est appelée sur l'instance nyPizzaStore.

- 3 **La méthode orderPizza() appelle d'abord la méthode createPizza() :**

```
Pizza pizza = createPizza("fromage");
```



À partir de là, les choses changent, car nous utilisons un facteur d'ingrédients y

Derrière
les scènes

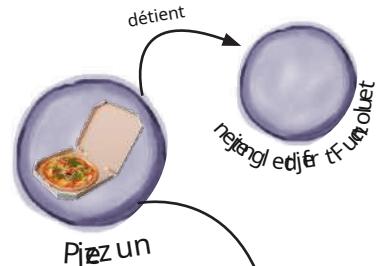


- 4 Lorsque la méthode `createPizza()` est appelée, c'est à ce moment-là que notre fabrique d'ingrédients entre en jeu :

L'usine d'ingrédients est choisie et instanciée dans le `PizzaStore`, puis transmise au constructeur de chaque pizza.

`Pizza pizza = nouvelle CheesePizza(nyIngredientFactory);`

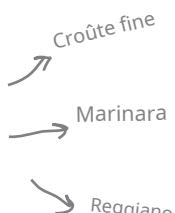
Crée une instance de Pizza qui est composée avec le Ingrédient de New York usine.



- 5 Ensuite, nous devons préparer la pizza. Une fois la méthode `prepare()` appelée, l'usine est invitée à préparer les ingrédients :

```
void préparer() {
    pâte = factory.createDough(); sauce =
    factory.createSauce(); fromage =
    factory.createCheese();
}
```

Pour la pizza d'Ethan, l'usine d'ingrédients de New York est utilisée, et nous obtenons donc les ingrédients de New York.



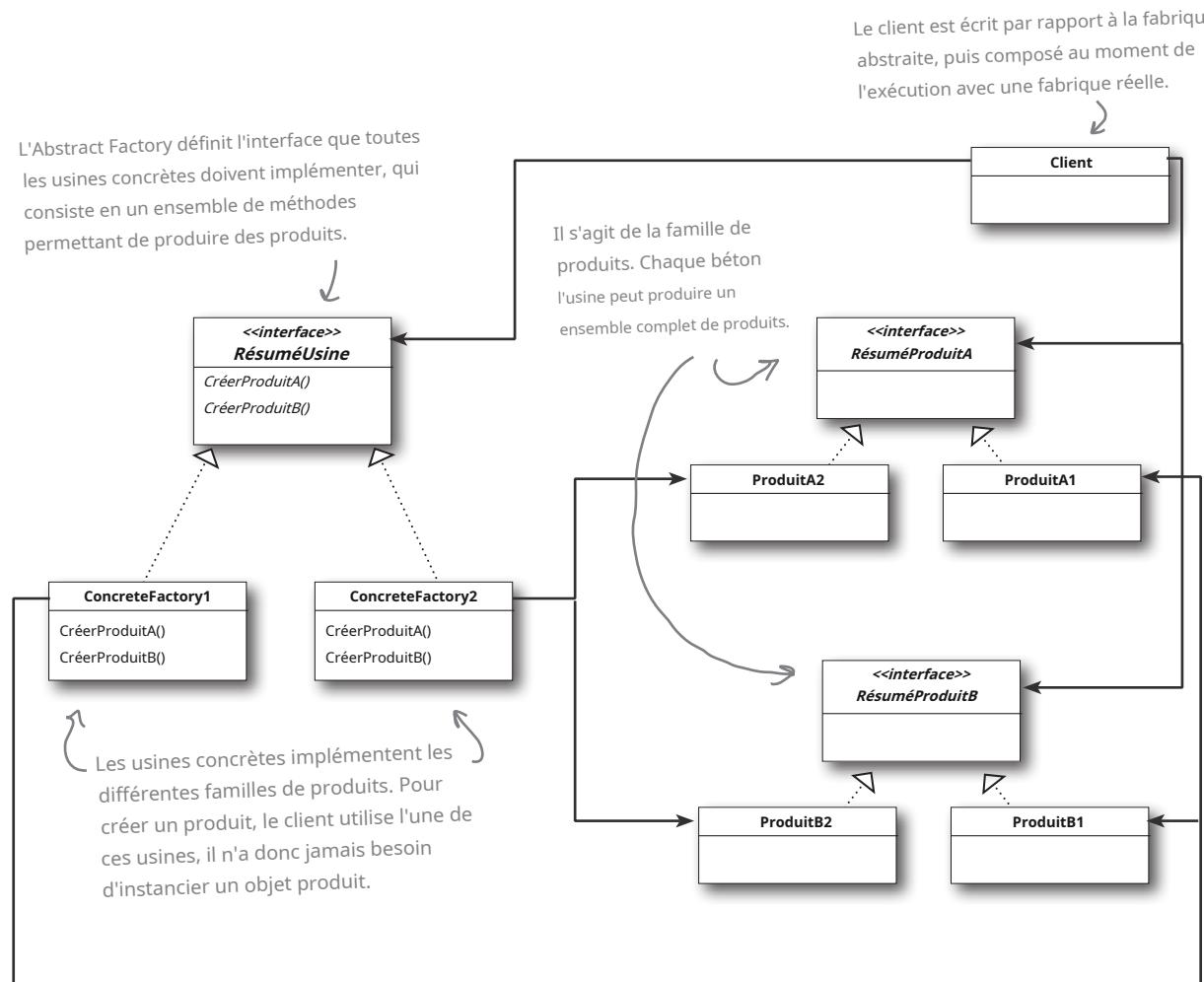
- 6 Enfin, nous avons la pizza préparée en main et la méthode `orderPizza()` cuit, coupe et met en boîte la pizza.

Modèle d'usine abstrait défini

Nous ajoutons un autre modèle d'usine à notre famille de modèles, qui nous permet de créer des familles de produits. Voyons la définition officielle de ce modèle :

Le modèle d'usine abstrait fournit une interface pour créer des familles d'objets liés ou dépendants sans spécifier leurs classes concrètes.

Nous avons certainement vu qu'Abstract Factory permet à un client d'utiliser une interface abstraite pour créer un ensemble de produits connexes sans connaître (ou se soucier) des produits concrets qui sont réellement produits. De cette façon, le client est découpé de toutes les spécificités des produits concrets. Regardons le diagramme de classes pour voir comment tout cela tient ensemble :

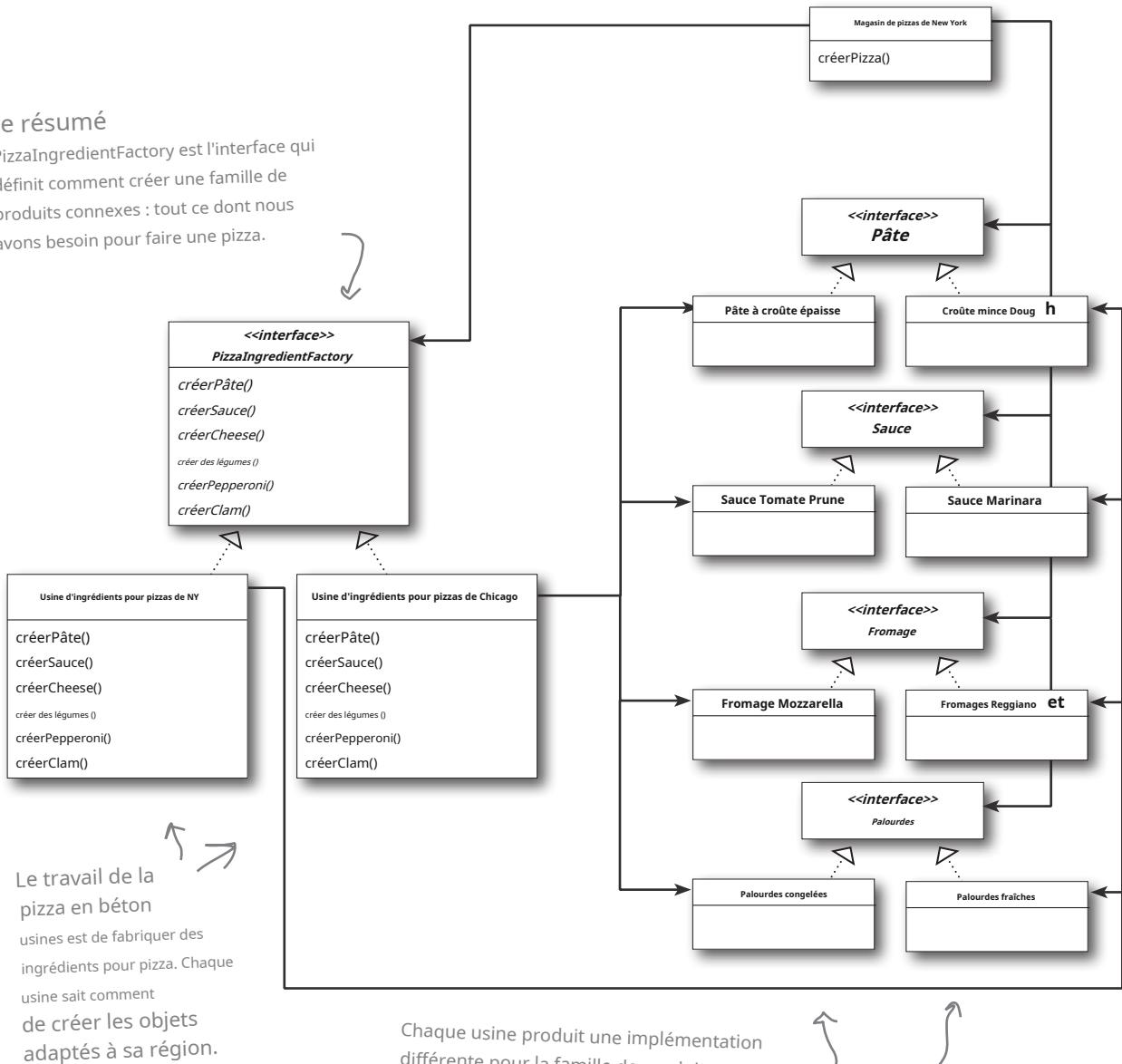


C'est un diagramme de classe assez compliqué ; regardons-le dans son ensemble en termes de notre PizzaStore :

Les clients de l'Abstract Factory sont les deux des exemples de notre PizzaStore, NYPizzaStore et Magasin de pizzas ChicagoStyle.

Le résumé

PizzaIngredientFactory est l'interface qui définit comment créer une famille de produits connexes : tout ce dont nous avons besoin pour faire une pizza.



Chaque usine produit une implémentation différente pour la famille de produits.



S'agit-il d'une méthode d'usine qui se cache à l'intérieur de l'usine abstraite ?

Bonne prise ! Oui, souvent les méthodes d'une fabrique abstraite sont implémentées en tant que méthodes de fabrique. C'est logique, non ? Le travail d'une fabrique abstraite est de définir une interface pour créer un ensemble de produits. Chaque méthode de cette interface est responsable de la création d'un produit concret, et nous implementons une sous-classe de la fabrique abstraite pour fournir ces implementations. Ainsi, les méthodes de fabrique sont un moyen naturel d'implémenter vos méthodes de produit dans vos fabriques abstraites.



Les modèles dévoilés

Ceinterview de la semaine :

Méthode de fabrique et fabrique abstraite, l'une sur l'autre

Tête la première :Waouh, une interview avec deux patrons à la fois ! C'est une première pour nous.

Méthode d'usine :Ouais, je ne suis pas sûr d'aimer être mis dans le même panier qu'Abstract Factory, tu sais. Ce n'est pas parce que nous sommes tous les deux des patrons d'usine que nous ne devrions pas avoir nos propres interviews.

Tête la première :Ne soyez pas vexé, nous voulions vous interviewer ensemble afin de clarifier toute confusion pour les lecteurs quant à qui est qui. Vous avez des similitudes, et j'ai entendu dire que les gens vous confondent parfois.

Usine abstraite :C'est vrai, il y a eu des moments où l'on m'a pris pour Factory Method, et je sais que tu as eu des problèmes similaires, Factory Method. Nous sommes tous les deux très bons pour découpler les applications des implementations spécifiques ; nous le faisons juste de manière différente. Je peux donc comprendre pourquoi les gens peuvent parfois nous confondre.

Méthode d'usine :Eh bien, ça me fait toujours tiquer. Après tout, j'utilise des classes pour créer et vous utilisez des objets ; c'est totalement différent !

Tête la première :Pouvez-vous expliquer davantage cela, Factory Method ?

Méthode d'usine :Bien sûr. Abstract Factory et moi créons des objets, c'est notre travail. Mais je le fais par héritage...

Usine abstraite :...et je le fais à travers la composition d'objets.

Méthode d'usine :D'accord. Cela signifie donc que pour créer des objets à l'aide de la méthode Factory, vous devez étendre une classe et fournir une implémentation pour une méthode Factory.

Tête la première :Et cette méthode d'usine fait quoi ?

Méthode d'usine :Bien sûr, cela crée des objets ! Je veux dire, tout l'intérêt du modèle de méthode d'usine est que vous utilisez une sous-classe pour effectuer votre création à votre place. De cette façon, les clients n'ont besoin de connaître que le type abstrait qu'ils utilisent ; la sous-classe s'occupe du type concret. Donc, en d'autres termes, je garde les clients découplés des types concrets.

Usine abstraite :Et moi aussi, mais je le fais d'une manière différente.

Tête la première :Allez, Abstract Factory... tu as dit quelque chose à propos de la composition d'objets ?

Usine abstraite :Je propose un type abstrait pour créer une famille de produits. Les sous-classes de ce type définissent la manière dont ces produits sont fabriqués. Pour utiliser la fabrique, vous en instanciez une et la transmettez à un code écrit par rapport au type abstrait. Ainsi, comme pour la méthode de fabrique, mes clients sont découplés des produits concrets qu'ils utilisent.

Tête la première :Oh, je vois, donc un autre avantage est que vous regroupez un ensemble de produits connexes.

Usine abstraite :C'est exact.

Tête la première :Que se passe-t-il si vous devez étendre cet ensemble de produits connexes pour, par exemple, en ajouter un autre ? Cela ne nécessite-t-il pas de modifier votre interface ?

Usine abstraite :C'est vrai ; mon interface doit changer si de nouveaux produits sont ajoutés, ce que je sais que les gens n'aiment pas faire...

Méthode d'usine :<ricanement>

Usine abstraite :De quoi tu te moques, Factory Method ?

Méthode d'usine :Oh, allez, c'est un gros problème ! Changer votre interface signifie que vous devez entrer et changer l'interface de chaque sous-classe ! Cela semble représenter beaucoup de travail.

Usine abstraite :Oui, mais j'ai besoin d'une grande interface parce que j'ai l'habitude de créer des familles entières de produits. Vous ne créez qu'un seul produit, donc vous n'avez pas vraiment besoin d'une grande interface, vous avez juste besoin d'une méthode.

Tête la première :Abstract Factory, j'ai entendu dire que vous utilisez souvent des méthodes d'usine pour mettre en œuvre vos usines à béton ?

Usine abstraite :Oui, je l'avoue, mes usines de béton mettent souvent en œuvre une méthode d'usine pour créer leurs produits. Dans mon cas, elles sont utilisées uniquement pour créer des produits...

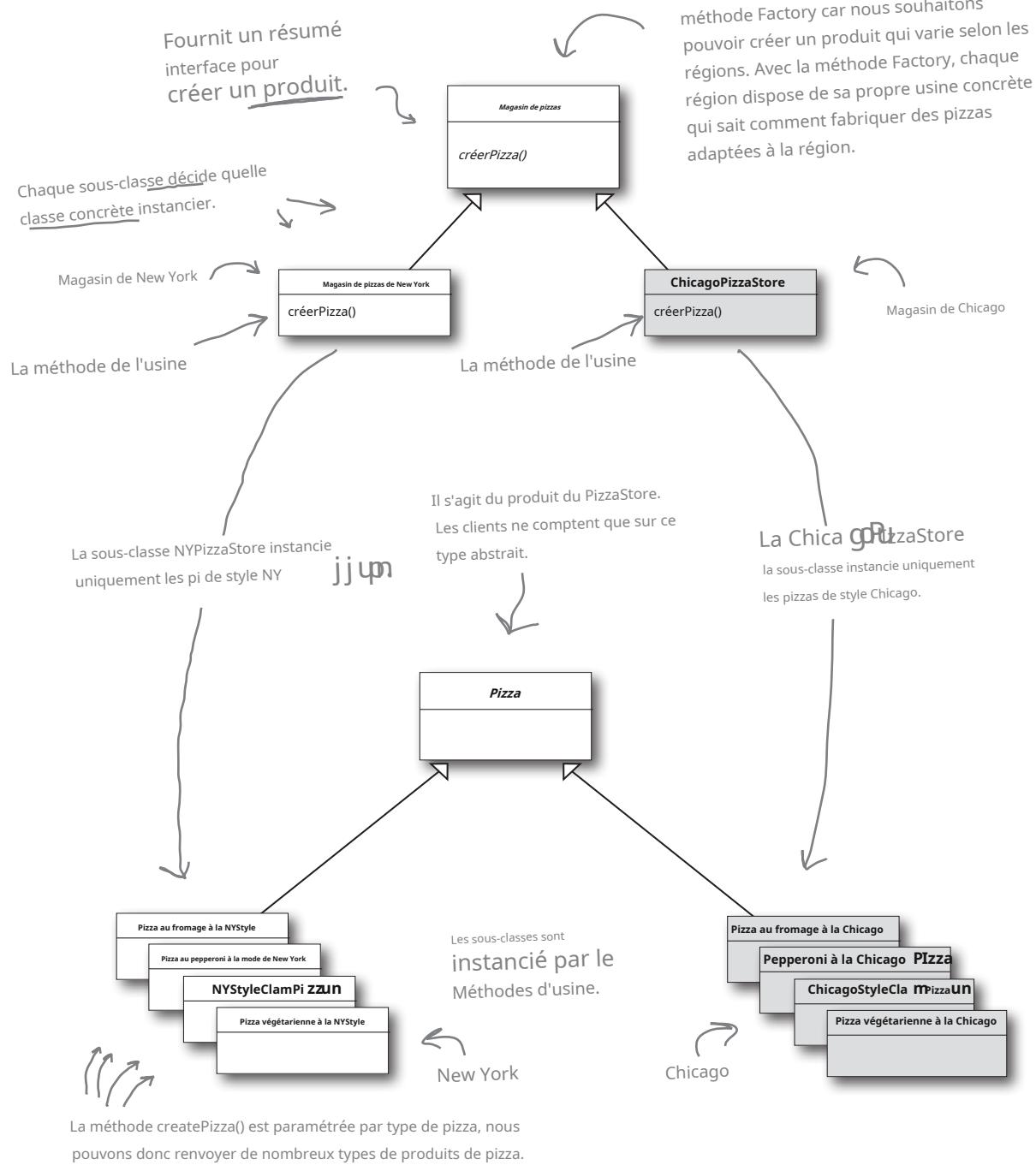
Méthode d'usine :...alors que dans mon cas, j'implémente généralement du code dans le créateur abstrait qui utilise les types concrets créés par les sous-classes.

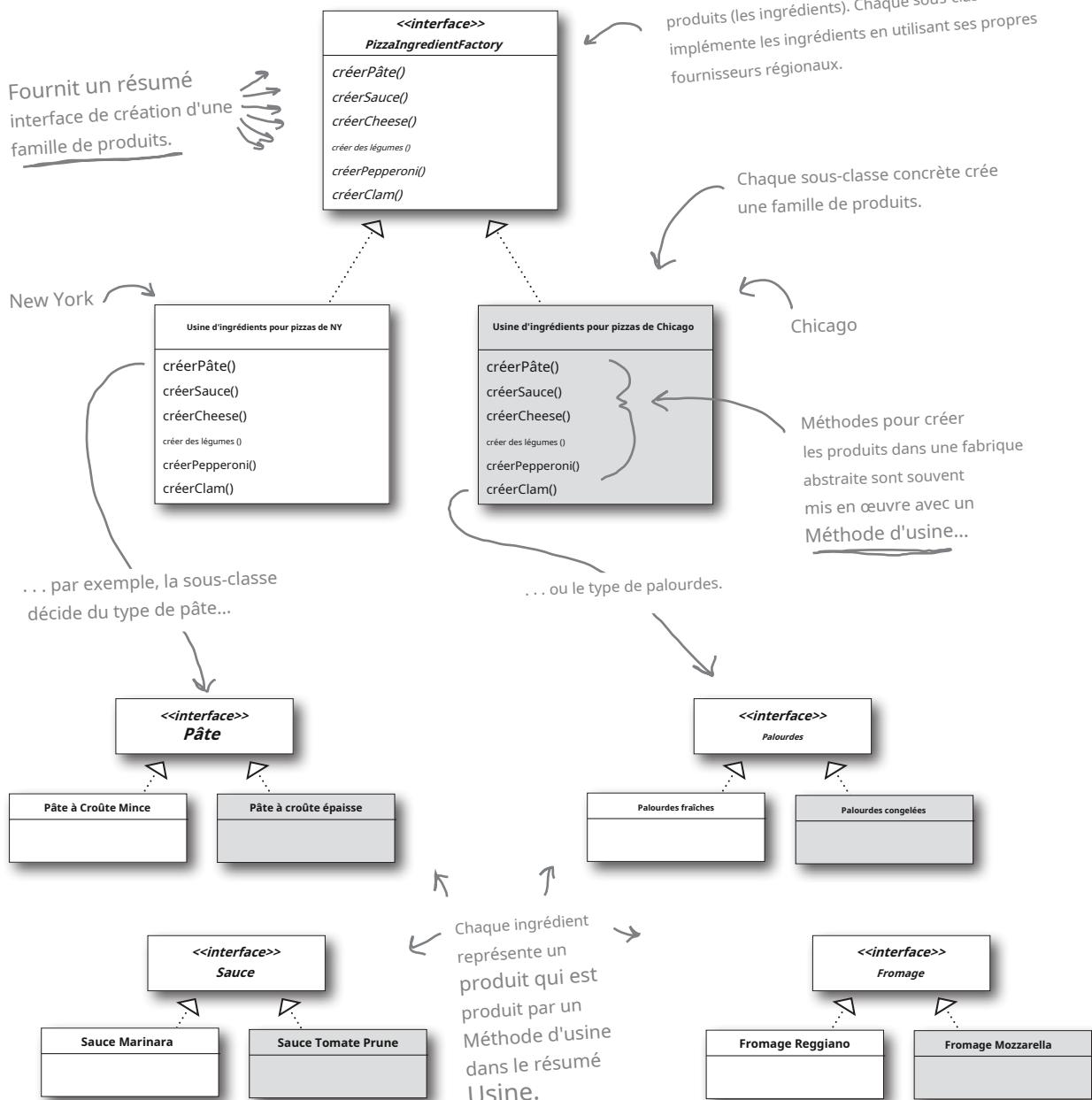
Tête la première :Il semble que vous soyez tous les deux bons dans ce que vous faites. Je suis sûr que les gens aiment avoir le choix ; après tout, les usines sont si utiles qu'ils voudront les utiliser dans toutes sortes de situations différentes. Vous encapsulez tous les deux la création d'objets pour garder les applications faiblement couplées et moins dépendantes des implémentations, ce qui est vraiment génial, que vous utilisez Factory Method ou Abstract Factory. Puis-je vous permettre à chacun un mot d'adieu ?

Usine abstraite :Merci. Souvenez-vous de moi, Abstract Factory, et faites appel à moi chaque fois que vous avez des familles de produits à créer et que vous voulez vous assurer que vos clients créent des produits qui vont ensemble.

Méthode d'usine :Et je suis la méthode Factory ; utilisez-moi pour découpler votre code client des classes concrètes que vous devez instancier, ou si vous ne connaissez pas à l'avance toutes les classes concrètes dont vous aurez besoin. Pour m'utiliser, sous-classez-moi simplement et implémentez ma méthode factory !

Comparaison entre la méthode Factory et la méthode Abstract Factory



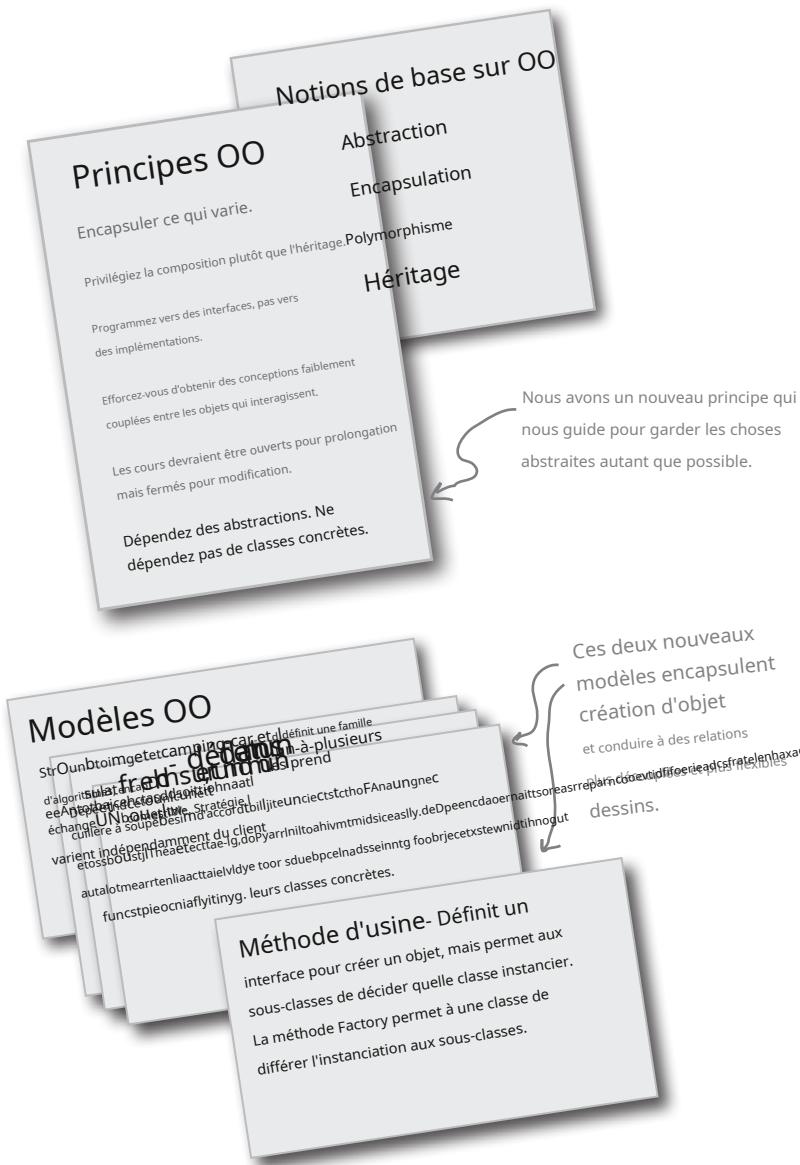


Les sous-classes de produits créent des ensembles parallèles de familles de produits. Nous avons ici une famille d'ingrédients de New York et une famille de Chicago.



Des outils pour votre boîte à outils de conception

Dans ce chapitre, nous avons ajouté deux outils supplémentaires à votre boîte à outils : Factory Method et Abstract Factory. Ces deux modèles encapsulent la création d'objets et vous permettent de découpler votre code des types concrets.



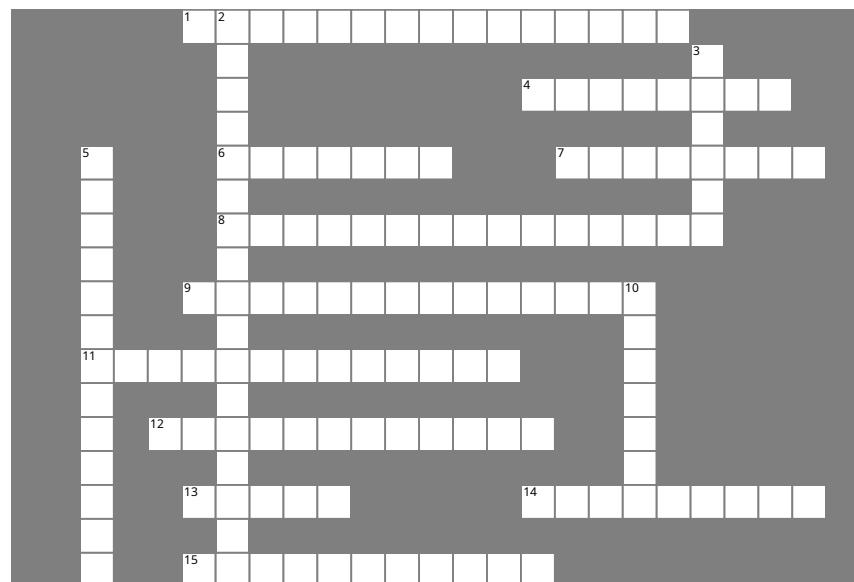
BULLET POINTS

- f Toutes les usines encapsulent la création d'objets.
- f Simple Factory, bien qu'il ne s'agisse pas d'un modèle de conception à part entière, est un moyen simple de découpler vos clients des classes concrètes.
- f La méthode Factory repose sur l'héritage : la création d'objets est déléguée à des sous-classes, qui implémentent la méthode Factory pour créer des objets.
- f Abstract Factory s'appuie sur la composition d'objets : la création d'objets est implémentée dans des méthodes exposées dans l'interface factory.
- f Tous les modèles d'usine favorisent le couplage lâche en réduisant la dépendance de votre application aux classes concrètes.
- f L'objectif de la méthode Factory est de permettre à une classe de différer l'instanciation à ses sous-classes.
- f L'objectif d'Abstract Factory est de créer des familles d'objets liés sans avoir à dépendre de leurs classes concrètes.
- f Le principe d'inversion de dépendance nous guide pour éviter les dépendances sur des types concrets et pour nous efforcer d'obtenir abstractions.
- f Les usines sont une technique puissante pour coder des abstractions, pas des classes concrètes.



Mots croisés sur les modèles de conception

Ce fut un long chapitre. Prenez une part de pizza et détendez-vous en faisant ces mots croisés ; tous les mots de solution proviennent de ce chapitre.



À TRAVERS

1. Dans la méthode Factory, chaque franchise est un _____.
4. Dans Factory Method, qui décide quelle classe instancier ?
6. Rôle de PizzaStore dans le modèle de méthode d'usine.
7. Toutes les pizzas de style new-yorkais utilisent ce type de fromage.
8. Dans Abstract Factory, chaque usine d'ingrédients est un _____.
9. Lorsque vous utilisez **nouveau**, vous programmez sur un _____.
11. `createPizza()` est un _____.
12. Joel aime ce genre de pizza.
13. Dans la méthode Factory, le PizzaStore et les pizzas concrètes dépendent tous de cette abstraction.
14. Lorsqu'une classe instancie un objet à partir d'une classe concrète, elle est _____ sur cet objet.
15. Tous les modèles d'usine nous permettent de _____ la création d'objets.

VERS LE BAS

2. Nous avons utilisé _____ dans Simple Factory et Abstract Factory, et l'héritage dans Factory Method.
3. Abstract Factory crée un _____ de produits.
5. Ce n'est pas un VRAI modèle d'usine, mais il est néanmoins pratique.
10. Ethan aime ce genre de pizza.



Sharpen your pencil

Solution

Nous avons terminé le NYPizzaStore ; il n'en reste plus que deux et nous serons prêts à franchiser ! Écrivez les implémentations de PizzaStore de style Chicago et de style Californien ici :

Ces deux magasins sont presque exactement comme le magasin de

New York... ils créent simplement différents types de pizzas.

```
classe publique ChicagoPizzaStore étend PizzaStore {
    Pizza protégée createPizza(élément de chaîne) {
        si (élément.equals("fromage")) {
            renvoyer une nouvelle ChicagoStyleCheesePizza(); }
        else if (item.equals("veggie")) {
            renvoyer une nouvelle ChicagoStyleVeggiePizza(); }
        else if (item.equals("clam")) {
            renvoyer une nouvelle ChicagoStyleClamPizza(); }
        else if (item.equals("pepperoni")) {
            renvoyer une nouvelle
            ChicagoStylePepperoniPizza(); } sinon renvoyer null;
    }
}
```

Pour la pizzeria de Chicago,
nous devons simplement
nous assurer de créer des
pizzas de style Chicago...

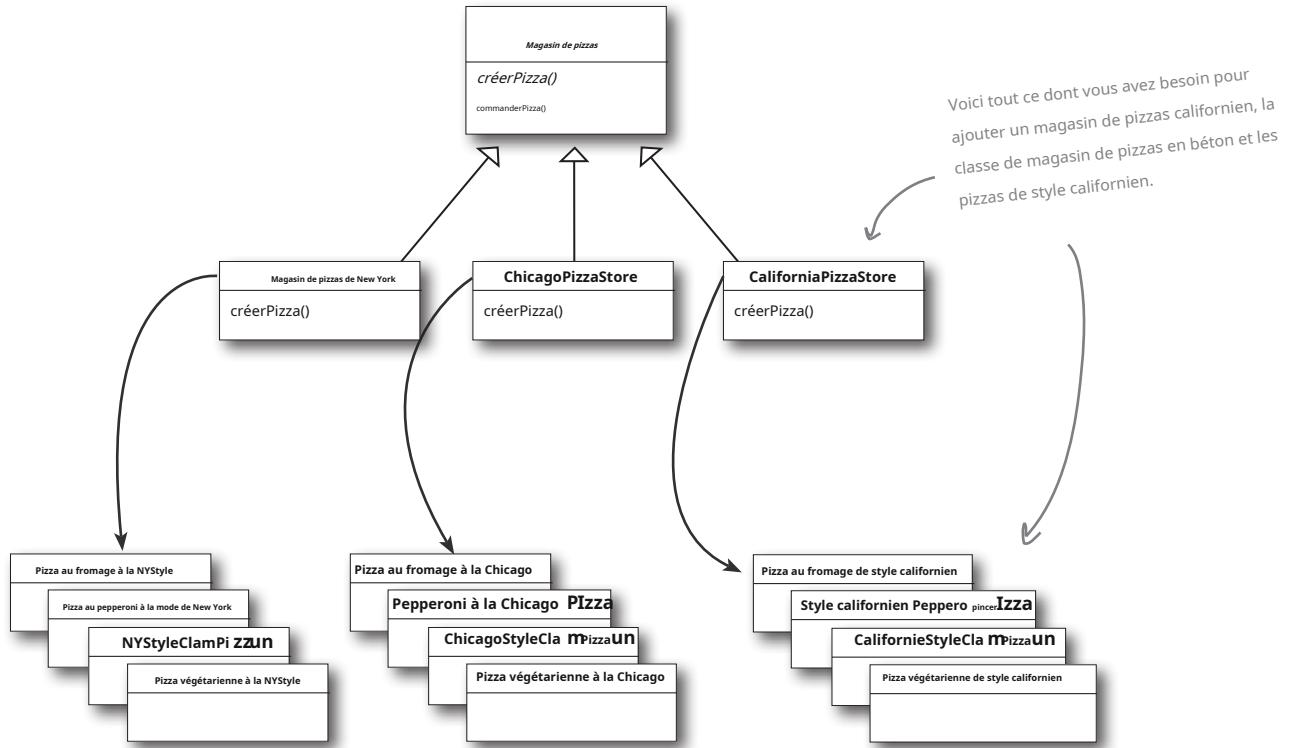
```
classe publique CaliforniaPizzaStore étend PizzaStore {
    Pizza protégée createPizza(élément de chaîne) {
        si (élément.equals("fromage")) {
            renvoyer une nouvelle CaliforniaStyleCheesePizza(); }
        else if (item.equals("veggie")) {
            renvoyer une nouvelle CaliforniaStyleVeggiePizza(); }
        else if (item.equals("clam")) {
            renvoyer une nouvelle CaliforniaStyleClamPizza(); }
        else if (item.equals("pepperoni")) {
            renvoyer une nouvelle CaliforniaStylePepperoniPizza(); }
        sinon renvoyer null;
    }
}
```

... et pour le magasin de pizzas
de Californie, nous créons des
pizzas de style californien.



Solution du puzzle de conception

Nous avons besoin d'un autre type de pizza pour ces Californiens fous (fous dans un *bien* bien sûr).
Dessinez un autre ensemble parallèle de classes dont vous auriez besoin pour ajouter une nouvelle région de Californie à notre PizzaStore.



Voici tout ce dont vous avez besoin pour ajouter un magasin de pizzas californien, la classe de magasin de pizzas en béton et les pizzas de style californien.

Bon, écrivez maintenant les cinq choses les plus idiotes que vous pouvez imaginer pour mettre sur une pizza. Ensuite, vous serez prêt à vous lancer dans la fabrication de pizzas en Californie !

Ici
sont nos
des suggestions...

Purée de pommes de terre à l'ail rôti

Sauce barbecue

Coeurs d'artichauts

M&M's

Cacahuètes



Sharpen your pencil Solution

Imaginons que vous n'avez jamais entendu parler d'une fabrique OO. Voici une version « très dépendante » de PizzaStore qui n'utilise pas de fabrique. Nous avons besoin que vous comptiez le nombre de classes de pizza concrètes dont dépend cette classe. Si vous ajoutez des pizzas de style californien à PizzaStore, de combien de classes dépendrait-elle alors ? Voici notre solution.

classe publique DependentPizzaStore {

```

public Pizza createPizza(Style de chaîne, Type de chaîne) {
    Pizza pizza = null; si
    (style.equals("NY")) {
        si (type.equals("fromage")) {
            pizza = nouvelle NYStyleCheesePizza(); }
        else if (type.equals("végétarien")) {
            pizza = nouvelle NYStyleVeggiePizza(); }
        sinon si (type.equals("clam")) {
            pizza = nouvelle NYStyleClamPizza(); }
        sinon si (type.equals("pepperoni")) {
            pizza = nouveau NYStylePepperoniPizza(); }
    }
    } sinon si (style.equals("Chicago")) {
        si (type.equals("fromage")) {
            pizza = nouvelle ChicagoStyleCheesePizza(); }
        else if (type.equals("végétarien")) {
            pizza = nouvelle ChicagoStyleVeggiePizza(); }
        else if (type.equals("clam")) {
            pizza = nouvelle ChicagoStyleClamPizza(); }
        sinon si (type.equals("pepperoni")) {
            pizza = nouvelle ChicagoStylePepperoniPizza(); }
    }
    } autre {
        System.out.println("Erreur : type de pizza non valide "); return
        null;
    }
    pizza.préparer();
    pizza.cuire();
    pizza.couper();
    boîte à pizza();
    retourner la pizza;
}
}

```

Gère tous les Pizzas à la new-yorkaise

Gère tous les Pizzas à la Chicago

Vous pouvez écrire
vos réponses ici :

8 nombre

12 numéro avec
La Californie aussi

Sharpen your pencil

Solution

Allez-y et écrivez la ChicagoPizzaIngredientFactory ; vous pouvez référencer les classes ci-dessous dans votre implémentation :

```

classe publique ChicagoPizzaIngredientFactory
implémente PizzaIngredientFactory
{

    public Pâte createDough() {
        renvoie un nouveau ThickCrustDough();
    }

    Sauce publique createSauce() {
        renvoie une nouvelle PlumTomatoSauce();
    }

    public Fromage createCheese() {
        renvoie un nouveau MozzarellaCheese();
    }

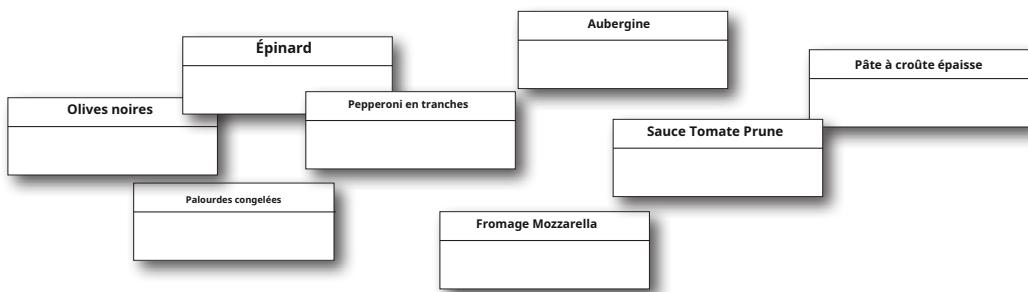
    public Veggies[] createVeggies() {
        Légumes veggies[] = { new BlackOlives(),
            nouveaux épinards(),
            nouvelle Aubergine() };

        remettre les légumes;
    }

    public Pepperoni createPepperoni() {
        renvoie un nouveau SlicedPepperoni();
    }

    palourdes publiques createClam() {
        renvoie un nouveau FrozenClams();
    }
}

```





Solution de mots croisés sur les modèles de conception

Ce fut un long chapitre. Prenez une part de pizza et détendez-vous en faisant ces mots croisés ; tous les mots de solution proviennent de ce chapitre. Voici la solution.

	¹ C	² O	N	C	R	E	T	E	C	R	E	U	N	T	O	R												
	B															³ F												
	J															S	Tu	B	C	L	UN	S	S					
	E															M												
⁵ S		⁶ C	R	E	U	N	T	O	R							⁷ R	E	G	G	je	U	N	N	O				
je		T														L												
M		⁸ C	O	N	C	R	E	T	E	F	U	N	C	T	O	R	Y											
P		O																										
L		⁹ je	M	P	L	E	M	E	N	T	U	N	T	je	O	¹⁰ N												
E		P														Y												
¹¹ F	U	N	C	T	O	R	Y	M	E	T	H	O	D			S												
UN		S														T												
C		¹² C	H	je	C	U	N	G	O	S	T	Y	L	E		Y												
T		T														L												
O		¹³ P	je	Z	Z	U	N									¹⁴ D	E	P	E	N	D	E	N	T				
R		O																										
Y		¹⁵ E	N	C	U	N	P	S	Tu	L	U	N	T	E														

5e modèle Singleton

* *Des objets uniques* *



Notre prochain arrêt est le modèle Singleton, notre ticket pour créer des objets uniques pour lesquels il n'existe qu'une seule instance, jamais. Vous pourriez

Soyez heureux de savoir que de tous les modèles, le Singleton est le plus simple en termes de diagramme de classes ; en fait, le diagramme ne contient qu'une seule classe ! Mais ne vous sentez pas trop à l'aise ; malgré sa simplicité du point de vue de la conception de classes, il va nécessiter une réflexion approfondie orientée objet dans sa mise en œuvre. Alors, mettez votre casquette de réflexion et allons-y.



Promoteur: A quoi ça sert ?

Gourou: Il existe de nombreux objets dont nous n'avons besoin que d'un seul : pools de threads, caches, boîtes de dialogue, objets qui gèrent les préférences et les paramètres de registre, objets utilisés pour la journalisation et objets qui agissent comme pilotes de périphériques pour des périphériques tels que des imprimantes et des cartes graphiques. En fait, pour bon nombre de ces types d'objets, si nous devions en instancier plusieurs, nous rencontrerions toutes sortes de problèmes tels qu'un comportement incorrect du programme, une surutilisation des ressources ou des résultats incohérents.

Promoteur: D'accord, il y a peut-être des classes qui ne devraient être instanciées qu'une seule fois, mais est-ce que j'ai besoin d'un chapitre entier pour cela ? Ne puis-je pas simplement le faire par convention ou par des variables globales ? Vous savez, comme en Java, je pourrais le faire avec une variable statique.

Gourou: À bien des égards, le modèle Singleton est une convention permettant de garantir qu'un seul et unique objet est instancié pour une classe donnée. Si vous en avez un meilleur, le monde aimeraient en entendre parler ; mais n'oubliez pas que, comme tous les modèles, le modèle Singleton est une méthode éprouvée pour garantir qu'un seul objet soit créé. Le modèle Singleton nous offre également un point d'accès global, tout comme une variable globale, mais sans les inconvénients.

Promoteur: Quels inconvénients ?

Gourou: Eh bien, voici un exemple : si vous affectez un objet à une variable globale, cet objet peut être créé au démarrage de votre application. N'est-ce pas ? Que se passe-t-il si cet objet nécessite beaucoup de ressources et que votre application ne l'utilise jamais ? Comme vous le verrez, avec le modèle Singleton, nous pouvons créer nos objets uniquement lorsqu'ils sont nécessaires.

Promoteur: Cela ne semble toujours pas si difficile.

Gourou: Si vous maîtrisez bien les variables et méthodes de classe statiques ainsi que les modificateurs d'accès, ce n'est pas le cas. Mais, dans les deux cas, il est intéressant de voir comment fonctionne un Singleton, et, aussi simple que cela puisse paraître, le code Singleton est difficile à obtenir correctement. Posez-vous simplement la question : comment empêcher linstanciation de plusieurs objets ? Ce n'est pas si évident, n'est-ce pas ?

Le petit célibataire

Un petit exercice socratique dans le style de La Petite Lisière

Comment créeriez-vous un objet unique ?

nouveau MonObjet();

Et si un autre objet voulait créer un MyObject ?

Oui bien sûr.

Pourrait-il à nouveau appeler new sur MyObject ?

Ainsi, tant que nous avons une classe, pouvons-nous toujours l'instancier une ou plusieurs fois ?

Oui. Enfin, seulement s'il s'agit d'un cours public.

Et sinon ?

Eh bien, s'il ne s'agit pas d'une classe publique, seules les classes du même package peuvent l'instancier. Mais elles peuvent toujours l'instancier plusieurs fois.

Hmm, intéressant.

Non, je n'y avais jamais pensé, mais je suppose que cela a du sens car c'est une définition juridique.

Saviez-vous que vous pouviez faire cela ?

```
publique MaClasse {  
  
    privé MaClasse() {}  
  
}
```

Qu'est-ce que ça veut dire?

Je suppose que c'est une classe qui ne peut pas être instanciée car elle possède un constructeur privé.

Eh bien, existe-t-il un objet qui pourrait utiliser le constructeur privé ?

Hmm, je pense que le code dans MyClass est le seul code qui pourrait l'appeler. Mais cela n'a pas beaucoup de sens.

Pourquoi pas ?

Parce que je devrais avoir une instance de la classe pour l'appeler, mais je ne peux pas avoir d'instance car aucune autre classe ne peut l'instancier. C'est le problème de l'œuf et de la poule : je peux utiliser le constructeur d'un objet de type MyClass, mais je ne peux jamais instancier cet objet car aucun autre objet ne peut utiliser « new MyClass() ».

Ok, c'était juste une idée.

Qu'est-ce que cela signifie?

MyClass est une classe avec une méthode statique. Nous pouvons appeler la méthode statique comme ceci :

MaClasse.getInstance();

```
publique MaClasse {  
  
    public static MaClasse getInstance() {}  
  
}
```

Pourquoi avez-vous utilisé MyClass au lieu d'un nom d'objet ?

Eh bien, getInstance() est une méthode statique ; en d'autres termes, c'est une méthode de CLASSE. Vous devez utiliser le nom de la classe pour référencer une méthode statique.

Très intéressant. Et si on mettait les choses ensemble ?

Waouh, tu peux certainement le faire.

Maintenant puis-je instancier une MyClass ?

```
publique MaClasse {  
  
    privé MaClasse() {}  
  
    public static MaClasse getInstance() {  
        renvoie une nouvelle MyClass();  
    }  
}
```

Alors, pouvez-vous maintenant penser à une deuxième façon d'instancier un objet ?

MaClasse.getInstance();

Pouvez-vous terminer le code afin qu'une seule instance de MyClass soit créée ?

Oui, je pense que oui...

(Vous trouverez le code sur la page suivante.)

Analyse de l'implémentation classique du modèle Singleton

```

Renommons
MaClasse vers Singleton.

classe publique Singleton {
    Singleton statique privé uniqueInstance;

    // autres variables d'instance utiles ici

    Singleton privé() {}

    public static Singleton getInstance() {
        si (uniqueInstance == null) {
            uniqueInstance = nouveau Singleton();
        }
        renvoyer une instance unique;
    }

    // autres méthodes utiles ici
}

```

Nous avons une variable statique pour contenir notre instance de la classe Singleton.

Notre constructeur est déclaré privé ; seulement Singleton peut instancier cette classe !

La méthode getInstance() nous donne un moyen d'instancier la classe et également de renvoyer une instance de celle-ci.

Bien sûr, Singleton est une classe normale ; elle possède d'autres variables d'instance et méthodes utiles.



Le code de près

uniqueInstance contient notre instance UNIQUE ; rappelez-vous, c'est une variable statique.

```

si (uniqueInstance == null) {
    uniqueInstance = nouveau Singleton();
}

renvoyer une instance unique;

```

Si uniqueInstance est nul, alors nous n'avons pas encore créé l'instance...

... et, s'il n'existe pas, nous instancions Singleton via son constructeur privé et l'assignons à uniqueInstance. Notez que si nous n'avons jamais besoin de l'instance, elle n'est jamais créée ; il s'agit d'une instantiation paresseuse.

Au moment où nous atteignons ce code, nous avons une instance et nous la renvoyons.

Si uniqueInstance n'était pas null, alors elle a été créée précédemment. Nous passons simplement à l'instruction return.



Les modèles dévoilés

Cette semaine entretien: Confessions d'une célibataire

Tête la première : Aujourd'hui, nous avons le plaisir de vous proposer une interview avec un objet Singleton. Pourquoi ne pas commencer par nous parler un peu de vous ?

Singleton: Eh bien, je suis totalement unique ; il n'y a qu'un seul exemplaire de moi !

Tête la première : Un ?

Singleton: Oui, un. Je suis basé sur le modèle Singleton, qui garantit qu'à tout moment, il n'existe qu'une seule instance de moi.

Tête la première : N'est-ce pas une sorte de gaspillage ? Quelqu'un a pris le temps de développer une classe à part entière et maintenant tout ce que nous pouvons en tirer, c'est un seul objet ?

Singleton: Pas du tout ! Le pouvoir réside dans UN. Imaginons que vous ayez un objet contenant des paramètres de registre. Vous ne voulez pas que plusieurs copies de cet objet et de ses valeurs circulent, car cela mènerait au chaos. En utilisant un objet comme moi, vous pouvez vous assurer que chaque objet de votre application utilise la même ressource globale.

Tête la première : Dites-nous en plus...

Singleton: Oh, je suis bon pour tout type de choses. Être célibataire a parfois ses avantages, tu sais. Je suis souvent habitué à gérer des pools de ressources, comme des pools de connexions ou de threads.

Tête la première : Et pourtant, tu n'es qu'un seul de ton espèce ? Ça semble solitaire.

Singleton: Comme je suis unique, je suis très occupé, mais ce serait bien si davantage de développeurs me connaissaient : de nombreux développeurs rencontrent des bugs parce qu'ils ont plusieurs copies d'objets qui circulent sans même en avoir conscience.

Tête la première : Alors, si on peut se permettre de se poser la question, comment savez-vous qu'il n'y a qu'un seul « vous » ? N'importe qui ne peut-il pas créer un « nouveau vous » avec un nouvel opérateur ?

Singleton: Non ! Je suis vraiment unique.

Tête la première : Eh bien, les développeurs jurent-ils de ne pas vous instancier plus d'une fois ?

Singleton: Bien sûr que non. À vrai dire... eh bien, cela devient un peu personnel mais... je n'ai pas de constructeur public.

Tête la première : PAS DE CONSTRUCTEUR PUBLIC ! Oh, pardon, pas de constructeur public ?

Singleton: C'est vrai. Mon constructeur est déclaré privé.

Tête la première : Comment ça marche ? Comment est-ce que tu arrives à être instancié ?

Singleton: Vous voyez, pour obtenir un objet Singleton, vous n'en créez pas un, vous demandez simplement une instance. Ma classe a donc une méthode statique appelée getInstance(). Appelez-la et j'apparaîtrai immédiatement, prêt à travailler. En fait, il se peut que j'aide déjà d'autres objets lorsque vous me le demandez.

Tête la première : Eh bien, M. Singleton, il semble y avoir beaucoup de choses sous vos couvertures pour que tout cela fonctionne. Merci de vous être révélé et nous espérons vous parler à nouveau bientôt !

La fabrique de chocolat

Tout le monde sait que toutes les usines de chocolat modernes sont équipées de chaudières à chocolat contrôlées par ordinateur. La fonction de la chaudière est de recueillir le chocolat et le lait, de les porter à ébullition, puis de les transmettre à l'étape suivante de fabrication des barres de chocolat.

Voici la classe de contrôleur pour la chaudière à chocolat de qualité industrielle de Choc-O-Holic, Inc. Regardez le code ; vous remarquerez qu'ils ont essayé d'être très prudents pour s'assurer que de mauvaises choses ne se produisent pas, comme vider 500 gallons de mélange non bouilli, ou remplir la chaudière alors qu'elle est déjà pleine, ou faire bouillir une chaudière vide !



```
classe publique ChocolateBoiler {
    booléen privé vide ;
    booléen privé bouilli;

    private ChocolateBoiler() {
        vide = vrai;
        bouilli = faux;
    }

    public void fill() {
        si (estVide()) {
            vide = faux;
            bouilli = faux;
            // remplir la chaudière avec un mélange lait/chocolat
        }
    }

    public void drain() {
        si (!estVide() && estBouilli()) {
            // égoutter le lait bouilli et le chocolat
            vide = true;
        }
    }

    public void ébullition() {
        si (!estVide() && !estBouilli()) {
            // porter le contenu à ébullition
            baked = true;
        }
    }

    public booléen est vide() {
        retourner vide;
    }

    public booléen isBoiled() {
        retourner bouilli;
    }
}
```

Ce code n'est déclenché que lorsque la chaudière est vide !

Pour remplir la chaudière, elle doit être vide, et, une fois pleine, nous mettons les drapeaux vide et bouilli.

Pour vider la chaudière, elle doit être pleine (non vide) et également bouillie. Une fois vidée, nous remettons le vide à la valeur réelle.

Pour faire bouillir le mélange, la chaudière doit être pleine et non déjà bouillie. Une fois qu'elle est bouillie, nous définissons l'indicateur d'ébullition sur vrai.



Choc-O-Holic a fait du bon travail pour s'assurer que de mauvaises choses ne se produisent pas, vous ne pensez pas ? Mais vous soupçonnez probablement que si deux instances de ChocolateBoiler se libèrent, de très mauvaises choses peuvent se produire.

Comment les choses pourraient-elles mal tourner si plusieurs instances de ChocolateBoiler sont créées dans une application ?



Sharpen your pencil

Pouvez-vous aider Choc-O-Holic à améliorer sa classe ChocolateBoiler en la transformant en Singleton ?

```
classe publique ChocolateBoiler {
```

booléen privé vide ;
booléen privé bouilli

Chaudière à chocolat ()

video = vrai:

bouilli = faux:

1

```
public void fill() {
```

```
    si (actVideo()) {
```

wide - form

bawilli = fawn

// remplir la chaudière avec un mélange lait/chocolat

1

1

// reste du code ChocolateBoiler

1

Modèle Singleton défini

Maintenant que vous avez en tête l'implémentation classique de Singleton, il est temps de vous asseoir, de savourer une barre de chocolat et de découvrir les points les plus subtils du modèle Singleton.

Commençons par la définition concise du modèle :

Le modèle Singleton garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à celle-ci.

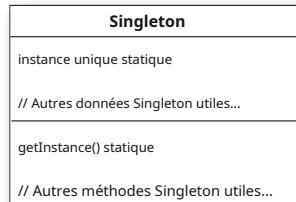
Pas de grande surprise ici. Mais analysons cela un peu plus en détail :

❖ Que se passe-t-il réellement ici ? Nous prenons une classe et la laissons gérer une seule instance d'elle-même. Nous empêchons également toute autre classe de créer elle-même une nouvelle instance. Pour obtenir une instance, vous devez passer par la classe elle-même.

❖ Nous fournissons également un point d'accès global à l'instance : chaque fois que vous avez besoin d'une instance, interrogez simplement la classe et elle vous renverra l'instance unique. Comme vous l'avez vu, nous pouvons implémenter cela de manière à ce que le singleton soit créé de manière paresseuse, ce qui est particulièrement important pour les objets gourmands en ressources.

Ok, regardons le diagramme de classe :

La méthode `getInstance()` est statique, ce qui signifie qu'il s'agit d'une méthode de classe. Vous pouvez donc accéder facilement à cette méthode depuis n'importe quel endroit de votre code à l'aide de `Singleton.getInstance()`. C'est aussi simple que d'accéder à une variable globale, mais nous bénéficiions d'avantages tels que l'instanciation paresseuse du Singleton.



L'instance unique
la variable de classe contient notre seule et unique instance de Singleton.

Une classe implementant le modèle Singleton est plus qu'un singleton ; c'est une classe à usage général avec son propre ensemble de données et de méthodes.

Hershey, Pennsylvanie

~~Houston~~, nous avons un problème...

Il semble que la chaudière à chocolat nous ait laissé tomber ; malgré le fait que nous ayons amélioré le code en utilisant le modèle Singleton classique, la méthode `fill()` de la chaudière à chocolat a réussi à commencer à remplir la chaudière même si un lot de lait et de chocolat était déjà en ébullition ! Cela représente 500 gallons de lait (et de chocolat) renversés ! Que s'est-il passé ?



SOYEZ la JVM



Nous avons deux threads, chacun exécutant ce code. Votre travail consiste à jouer avec la JVM et à déterminer s'il existe un cas dans lequel deux threads peuvent obtenir la main sur des objets boiler différents. Astuce :

**il suffit vraiment de regarder le
séquence d'opérations dans le
méthode getInstance() et la valeur
de uniqueInstance pour voir
comment elles pourraient
chevauchement. Utilisez les aimants de**

code pour vous aider à étudier comment le
le code peut s'entrelacer pour créer deux objets chaudière.

```
Chaudière ChocolateBoiler =
    ChocolateBoiler.getInstance();
    chaudière.fill();
    chaudière.ébullition();
    chaudière.drain();
```

```
chaudière à chocolat statique publique
obtenirInstance() {
```

```
    si (uniqueInstance == null) {
```

```
        uniqueInsta nce =
            nouveau Cho cchaudière olate();
```

```
}
```

```
        retourne uniqueJe nposition;
```

```
}
```

Assurez-vous de vérifier votre réponse à la page 188 avant de continuer !

Fil
Un

Fil
Deux

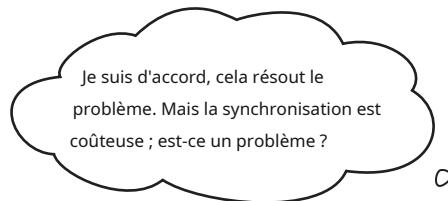
Valeur de
instance unique

Gérer le multithreading

Nos problèmes de multithreading sont presque trivialement résolus en faisant de `getInstance()` une méthode synchronisée :

```
classe publique Singleton {  
    Singleton statique privé uniqueInstance ;  
  
    // autres variables d'instance utiles ici  
  
    Singleton privé() {}  
  
    public statique  synchronisé  Singleton getInstance() {  
        si (uniqueInstance == null) {  
            uniqueInstance = nouveau Singleton();  
        }  
        renvoyer une instance unique ;  
    }  
  
    // autres méthodes utiles ici  
}
```

En ajoutant le mot-clé synchronisé à `getInstance()`, nous forçons chaque thread à attendre son tour avant de pouvoir entrer dans la méthode. Autrement dit, deux threads ne peuvent pas entrer dans la méthode en même temps.



Bonne remarque, et c'est en fait un peu pire que ce que vous dites : la seule synchronisation temporelle pertinente est la première fois que nous utilisons cette méthode. En d'autres termes, une fois que nous avons défini la variable `uniqueInstance` sur une instance de `Singleton`, nous n'avons plus besoin de synchroniser cette méthode. Après la première fois, la synchronisation est une surcharge totalement inutile !

Pouvons-nous améliorer le multithreading ?

Pour la plupart des applications Java, nous devons évidemment nous assurer que le Singleton fonctionne en présence de plusieurs threads. Mais la synchronisation de la méthode getInstance() est coûteuse, alors que faire ?

Eh bien, nous avons quelques options...

1. Ne faites rien si les performances de getInstance() ne sont pas critiques pour votre application.

C'est vrai ; si l'appel de la méthode getInstance() n'entraîne pas de surcharge importante pour votre application, oubliez-la. La synchronisation de getInstance() est simple et efficace. Gardez simplement à l'esprit que la synchronisation d'une méthode peut réduire les performances d'un facteur 100. Par conséquent, si une partie de votre code à fort trafic commence à utiliser getInstance(), vous devrez peut-être reconsidérer votre décision.

2. Passez à une instance créée avec empreusement plutôt qu'à une instance créée avec paresse.

Si votre application crée et utilise toujours une instance du Singleton, ou si la surcharge liée aux aspects de création et d'exécution du Singleton n'est pas onéreuse, vous souhaiterez peut-être créer votre Singleton avec empreusement, comme ceci :

```
classe publique Singleton {  
    Singleton statique privé uniqueInstance = new Singleton();  
  
    Singleton privé() {}  
  
    public static Singleton getInstance() {  
        renvoyer une instance unique ;  
    }  
}
```

Allez-y et créez une instance de Singleton dans un initialiseur statique. Ce code est garanti thread-safe !

Nous avons déjà une instance, il suffit donc de la renvoyer.

En utilisant cette approche, nous nous appuyons sur la JVM pour créer l'instance unique du Singleton lors du chargement de la classe. La JVM garantit que l'instance sera créée avant qu'un thread n'accède à la variable uniqueInstance statique.

3. Utilisez le « verrouillage à double vérification » pour réduire l'utilisation de la synchronisation dans getInstance().

Avec le verrouillage à double vérification, nous vérifions d'abord si une instance est créée, et si ce n'est pas le cas, ALORS nous synchronisons. De cette façon, nous synchronisons uniquement la première fois, exactement ce que nous voulons.

Regardons le code :

```
classe publique Singleton {  
    privé     volatile *Singleton statique uniqueInstance;  
  
    Singleton privé() {}  
  
    public static Singleton getInstance() {  
        si (uniqueInstance == null) {  
            synchronisé (Singleton.class) {  
                si (uniqueInstance == null) {  
                    uniqueInstance = nouveau Singleton();  
                }  
            }  
        }  
        renvoyer une instance unique;  
    }  
}
```

Recherchez une instance et s'il n'y en a pas, entrez un bloc synchronisé.

Notez que nous ne synchronisons que la première fois !

Une fois dans le bloc, vérifiez à nouveau et s'il est toujours nul, créez une instance.

* Le mot clé volatile garantit que plusieurs threads gèrent correctement la variable uniqueInstance lorsqu'elle est initialisée sur l'instance Singleton.

Si les performances sont un problème dans votre utilisation de la méthode getInstance(), cette méthode d'implémentation du Singleton peut réduire considérablement la surcharge.



Le verrouillage doublement vérifié ne fonctionne pas dans Java 1.4 ou une version antérieure !

Si pour une raison quelconque vous utilisez une ancienne version de Java, malheureusement, dans la version Java 1.4 et antérieure, de nombreuses JVM contiennent des implémentations de

le mot clé volatile qui permet une synchronisation incorrecte pour le verrouillage à double vérification. Si vous devez utiliser une JVM antérieure à Java 5, envisagez d'autres méthodes d'implémentation de votre Singleton.

Pendant ce temps, de retour à la Chocolaterie...

Pendant que nous étions en train de diagnostiquer les problèmes de multithreading, la chaudière à chocolat a été nettoyée et est prête à fonctionner. Mais d'abord, nous devons résoudre les problèmes de multithreading. Nous avons quelques solutions à portée de main, chacune avec des compromis différents, alors quelle solution allons-nous utiliser ?



Sharpen your pencil

Pour chaque solution, décrivez son applicabilité au problème de correction du code Chocolate Boiler :

Synchronisez la méthode getInstance() :

Utiliser l'instanciation anticipée :

Verrouillage doublement vérifié :

Félicitations !

À ce stade, la Chocolate Factory est un client satisfait et Choc-O-Holic était heureux d'avoir une certaine expertise appliquée à son code de chaudière. Quelle que soit la solution multithread que vous avez appliquée, la chaudière devrait être en bon état et ne plus avoir d'incidents. Félicitations, non seulement vous avez réussi à échapper à 500 livres de chocolat chaud dans ce chapitre, mais vous avez également surmonté tous les problèmes potentiels du modèle Singleton.

there are no Dumb Questions

Q: Pour un modèle aussi simple composé d'une seule classe, Singleton semble certainement avoir quelques problèmes.

UN: Eh bien, nous vous avions prévenu dès le départ ! Mais ne vous laissez pas décourager par les problèmes : même si l'implémentation correcte des Singletons peut être délicate, après avoir lu ce chapitre, vous êtes désormais bien informé sur les techniques de création de Singletons et vous devriez les utiliser partout où vous avez besoin de contrôler le nombre d'instances que vous créez.

Q: Ne puis-je pas simplement créer une classe dans laquelle toutes les méthodes et variables sont définies comme statiques ? Cela ne serait-il pas la même chose qu'un Singleton ?

UN: Oui, si votre classe est autonome et ne dépend pas d'une initialisation complexe. Cependant, en raison de la façon dont les initialisations statiques sont gérées en Java, cela peut devenir très compliqué, surtout si plusieurs classes sont impliquées. Souvent, ce scénario peut entraîner des bugs subtils et difficiles à trouver impliquant l'ordre d'initialisation. À moins qu'il n'y ait un besoin impérieux d'implémenter votre « singleton » de cette façon, il est de loin préférable de rester dans le monde des objets.

Q: Qu'en est-il des chargeurs de classes ? J'ai entendu dire qu'il y avait une chance que deux chargeurs de classes puissent chacun se retrouver avec leur propre instance de Singleton.

UN: Oui, c'est vrai, car chaque chargeur de classe définit un espace de noms. Si vous avez deux chargeurs de classe ou plus, vous pouvez charger la même classe plusieurs fois (une fois dans chaque chargeur de classe). Maintenant, si cette classe se trouve être un Singleton, alors comme nous avons plus d'une version de la classe, nous avons également plus d'une instance de Singleton. Donc, si vous utilisez plusieurs chargeurs de classe et Singletons, soyez prudent. Une façon de contourner ce problème est de spécifier vous-même le chargeur de classe.

Q: Et la réflexion, et la sérialisation/désérialisation ?

UN: Oui, la réflexion et la sérialisation/désérialisation peuvent également poser des problèmes avec les singletons. Si vous êtes un utilisateur Java avancé utilisant la réflexion, la sérialisation et la désérialisation, vous devrez garder cela à l'esprit.

Q: Nous avons évoqué précédemment le principe de couplage faible. Un Singleton ne viole-t-il pas ce principe ? Après tout, chaque objet de notre code qui dépend du Singleton sera étroitement couplé à cet objet très spécifique.

UN: Ouh là, c'est en fait une critique courante du modèle Singleton. Le principe de couplage faible dit qu'il faut « s'efforcer d'obtenir des conceptions faiblement couplées entre les objets qui interagissent ». Il est facile pour les Singletons de violer ce principe : si vous apportez une modification au Singleton, vous devrez probablement apporter une modification à chaque objet qui lui est connecté.

Q: On m'a toujours appris qu'une classe ne doit faire qu'une seule chose et une seule chose seulement. Le fait qu'une classe fasse deux choses est considéré comme une mauvaise conception OO. Un Singleton ne viole-t-il pas également cela ?

UN: Vous feriez référence au principe de responsabilité unique, et oui, vous avez raison : le singleton est responsable non seulement de la gestion de sa seule instance (et de la fourniture accès global), mais aussi pour son rôle principal dans votre application. On pourrait donc certainement dire qu'elle assume deux responsabilités. Néanmoins, il n'est pas difficile de voir qu'il est utile qu'une classe gère sa propre instance ; cela simplifie certainement la conception globale. De plus, de nombreux développeurs connaissent le modèle Singleton car il est largement utilisé. Cela dit, certains développeurs ressentent le besoin d'abstraire la fonctionnalité Singleton.

Q: Je voulais sous-classer mon code Singleton, mais j'ai rencontré des problèmes. Est-il possible de sous-classer un Singleton ?

UN: L'un des problèmes de la sous-classification d'un Singleton est que le constructeur est privé. Vous ne pouvez pas étendre une classe avec un constructeur privé. La première chose à faire est donc de modifier votre constructeur pour qu'il soit public ou protégé. Mais ce n'est plus vraiment un Singleton, car d'autres classes peuvent l'instancier.

Si vous modifiez votre constructeur, un autre problème se pose. L'implémentation de Singleton est basée sur une variable statique, donc si vous faites une sous-classe simple, toutes vos classes dérivées partageront la même variable d'instance. Ce n'est probablement pas ce que vous aviez en tête. Ainsi, pour que la sous-classe fonctionne, il est nécessaire d'implémenter une sorte de registre dans la classe de base.

Mais que gagnez-vous réellement à sous-classer un Singleton ? Comme la plupart des modèles, Singleton n'est pas nécessairement destiné à être une solution pouvant s'intégrer dans une bibliothèque. De plus, le code Singleton est facile à ajouter à une classe existante. Enfin, si vous utilisez un grand nombre de Singletons dans votre application, vous devez examiner attentivement votre conception. Les Singletons sont censés être utilisés avec parcimonie.

Q: Je ne comprends toujours pas totalement pourquoi les variables globales sont pires qu'un Singleton.

UN: En Java, les variables globales sont essentiellement des références statiques à des objets. L'utilisation de variables globales de cette manière présente quelques inconvénients. Nous en avons déjà mentionné un : le problème de l'instanciation paresseuse ou impatiente. Mais nous devons garder à l'esprit l'objectif du modèle : garantir qu'une seule instance d'une classe existe et fournir un accès global. Une variable globale peut fournir le second, mais pas le premier. Les variables globales ont également tendance à encourager les développeurs à polluer l'espace de noms avec de nombreuses références globales à de petits objets. Les singletons n'encouragent pas cela de la même manière, mais peuvent néanmoins être utilisés à mauvais escient.



Ah, bonne idée !

La plupart des problèmes que nous avons évoqués (s'inquiéter de la synchronisation, des problèmes de chargement de classe, de réflexion et de sérialisation/désérialisation) peuvent tous être résolus en utilisant une énumération pour créer votre Singleton. Voici comment procéder :

```
énumération publique Singleton {
    UNIQUE_INSTANCE;
    // d'autres champs utiles ici
}
classe publique SingletonClient {
    public static void main(String[] args) {
        Singleton singleton = Singleton.UNIQUE_INSTANCE; //
        utiliser le singleton ici
    }
}
```

Oui, c'est tout ce qu'il y a à faire. Le Singleton le plus simple de tous les temps, n'est-ce pas ? Maintenant, vous vous demandez peut-être pourquoi nous avons fait tout cela plus tôt avec la création d'une classe Singleton avec une méthode `getInstance()` puis la synchronisation, etc. Nous l'avons fait pour que vous compreniez vraiment comment fonctionne Singleton. Maintenant que vous le savez, vous pouvez utiliser enum chaque fois que vous avez besoin d'un Singleton, et toujours être capable de réussir cet entretien Java si la question se pose : « Comment implémenter un Singleton *sans* en utilisant enum ? »

Et à l'époque, quand nous devions marcher jusqu'à l'école, en montée, dans la neige, dans les deux sens, Java n'avait pas d'énumérations.



Pouvez-vous retravailler Choc-O-Holic pour utiliser une énumération ? Essayez.



Des outils pour votre boîte à outils de conception

Vous avez maintenant ajouté un autre modèle à votre boîte à outils. Singleton vous offre une autre méthode de création d'objets, dans ce cas, des objets uniques.

Principes OO

- Encapsuler ce qui varie.
- Privilégiez la composition plutôt que l'héritage.
- Programmez vers des interfaces, pas vers des implémentations.
- Efforcez-vous d'obtenir des conceptions faiblement couplées entre les objets qui interagissent.
- Les cours devraient être ouverts pour prolongation mais fermés pour modification.
- Dépendez des abstractions. Ne dépendez pas de classes concrètes.

Notions de base sur OO

- Abstraction
- Encapsulation
- Héritage

Modèles OO

- Le patron de conception **Singleton** définit une famille d'algorithmes, - définir c'est un un à plusieurs interchangables. Il fournit une interface pour hériter et des moyens pour varier individuellement les méthodes. Il est utile pour éviter de créer plusieurs instances d'un même objet. Y appelle également un singleton de de DP se rapporte au singleton et à la sécurité des personnes. Il permet une classe à différencier l'instanciation des sous-classes.

Comme vous l'avez vu, malgré son apparente simplicité, l'implémentation de Singleton implique de nombreux détails. Après avoir lu ce chapitre, vous êtes prêt à utiliser Singleton dans la nature.



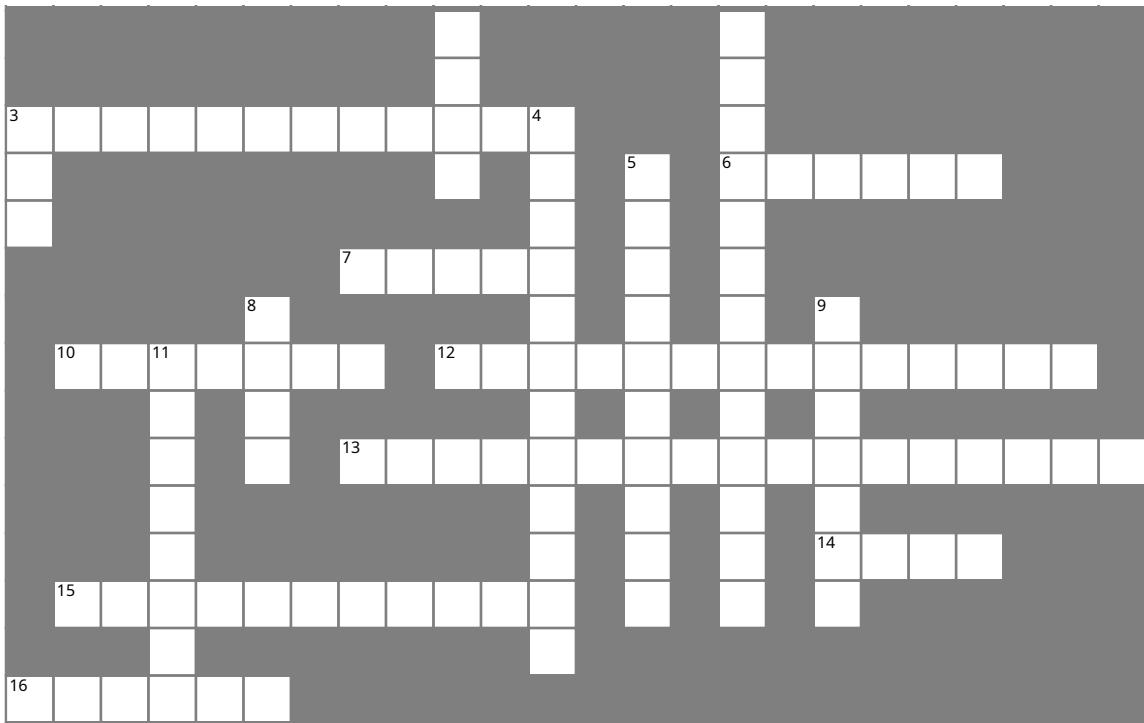
BULLET POINTS

- f Le modèle Singleton garantit que vous n'avez qu'une seule instance d'une classe dans votre application.
- f Le modèle Singleton fournit également un point d'accès global à cette instance.
- f Implémentation de Java du modèle Singleton utilise un constructeur privé, un statique méthode combinée avec une variable statique.
- f Examinez vos contraintes de performances et de ressources et choisissez soigneusement un Singleton approprié mise en œuvre pour applications multithread (et nous devrions considérer toutes les applications comme multithread !).
- f Attention au verrouillage à double contrôle implémentation ; elle n'est pas thread-safe dans les versions antérieures à Java 5.
- f Soyez prudent si vous utilisez plusieurs chargeurs de classes ; cela pourrait contrecarrer l'implémentation Singleton et entraîner plusieurs instances.
- f Vous pouvez utiliser les énumérations de Java pour simplifier votre implémentation Singleton.



Mots croisés sur les modèles de conception

Asseyez-vous, ouvrez cette caisse de chocolat qui vous a été envoyée pour résoudre le problème du multithreading et prenez un peu de temps libre pour travailler dessus.
 pediasuəs oudp zzl ; tout deti olution moids sonde mlth est c apter.



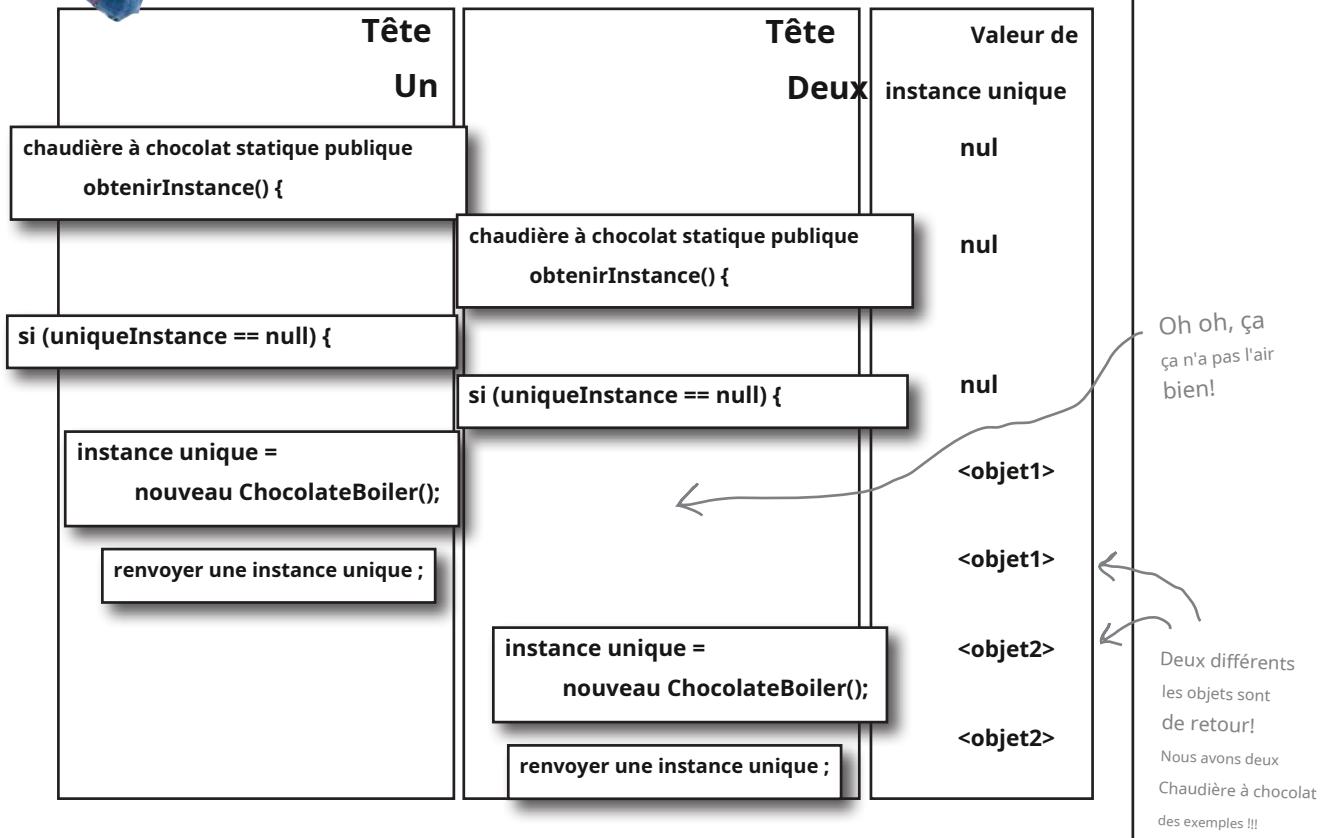
À TRAVERS

3. Entreprise qui produit des chaudières.
6. Une implémentation incorrecte a provoqué ce débordement.
7. Le modèle Singleton en a un.
10. Pour vaincre totalement le nouveau constructeur, nous devons déclarer le constructeur _____.
12. L'implémentation classique ne gère pas cela.
13. Singleton fournit une instance unique et _____ (trois mots).
14. Un moyen simple de créer des singletons en Java.
15. Le Singleton était gêné de ne pas avoir de relations publiques _____.
16. Un singleton est une classe qui gère une instance de _____.

VERS LE BAS

1. Ajouté au chocolat dans la chaudière.
2. Approche multithread défectueuse si vous n'utilisez pas Java 5 ou une version ultérieure (deux mots).
3. C'était « unique en son genre ».
4. Plusieurs _____ peuvent causer des problèmes (deux mots).
5. Si vous n'avez pas besoin de vous soucier de l'instanciation paresseuse, vous pouvez créer votre instance _____.
8. Un avantage par rapport aux variables globales : création de _____.
9. Capitale du chocolat aux États-Unis.
11. Singleton garantit qu'un seul d'entre eux existe.

SOYEZ la solution JVM





Pouvez-vous aider Choc-O-Holic à améliorer sa classe ChocolateBoiler en la transformant en Singleton ?

```

classe publique ChocolateBoiler {
    booléen privé vide ;
    booléen privé bouilli;

    ChocolateBoiler statique privé uniqueInstance ;

    ChocolateBoiler privé() {
        vide = vrai;
        bouilli = faux;
    }

    public static ChocolateBoiler getInstance() {
        si (uniqueInstance == null) {
            uniqueInstance = nouvelle ChocolateBoiler();
        }
        renvoyer une instance unique ;
    }

public void fill() {
    si (estVide()) {
        vide = faux;
        bouilli = faux;
        // remplir la chaudière avec un mélange lait/chocolat
    }
    }
    // reste du code ChocolateBoiler...
}

```



Sharpen your pencil Solution

Pour chaque solution, décrivez son applicabilité au problème de correction du code Chocolate Boiler :

Synchronisez la méthode getInstance() :

Une technique simple qui fonctionne à coup sûr. Nous ne semblons pas avoir de

problèmes de performances avec la chaudière à chocolat, ce serait donc un bon choix.

Utiliser l'instanciation anticipée :

Nous allons toujours instancier la chaudière à chocolat dans notre code, donc l'initialisation statique de

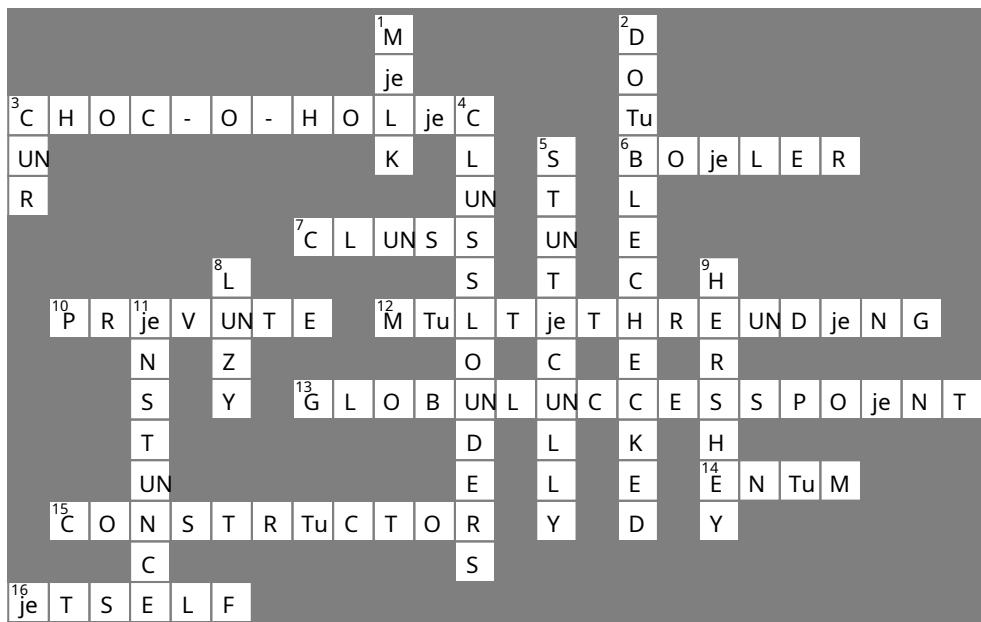
l'instance ne poserait aucun problème. Cette solution fonctionnerait aussi bien que la méthode synchronisée,

bien qu'elle soit peut-être moins évidente pour un développeur familier avec le modèle standard.

Verrouillage doublement vérifié :

Étant donné que nous n'avons aucun problème de performances, le verrouillage à double vérification semble

excessif. De plus, nous devons nous assurer que nous exécutons au moins Java 5.



Conception Motifs Mots croisés Solution



Invocation encapsulante

Ces boîtes de dépôt ultra-secrètes ont révolutionné l'industrie de l'espionnage. Je dépose simplement ma demande et les gens disparaissent, les gouvernements changent du jour au lendemain et mon nettoyage à sec est fait. Je n'ai pas à me soucier du moment, du lieu ou de la date.

comment; ça arrive tout simplement!



Dans ce chapitre, nous amenons l'encapsulation à un tout autre niveau : nous allons encapsuler l'invocation de méthode. C'est vrai, par

En encapsulant l'invocation de méthode, nous pouvons cristalliser des parties de calcul de sorte que l'objet qui invoque le calcul n'a pas à se soucier de la manière de faire les choses, il utilise simplement notre méthode cristallisée pour le faire. Nous pouvons également faire des choses incroyablement intelligentes avec ces invocations de méthodes encapsulées, comme les enregistrer pour la journalisation ou les réutiliser pour implémenter la fonctionnalité d'annulation dans notre code.



**Home Automation or Bust, Inc. 1221
Industrial Avenue, Suite 2000 Future
City, IL 62914**

Salutations!

J'ai récemment reçu une démonstration et un briefing de Johnny Hurricane, PDG de Weather-O-Rama, sur leur nouvelle station météorologique extensible. Je dois dire que j'ai été tellement impressionné par l'architecture logicielle que j'aimerais vous demander de concevoir l'API de notre nouvelle télécommande domotique. En échange de vos services, nous serions heureux de vous récompenser généreusement avec des options sur actions de Home Automation or Bust, Inc.

Vous devriez déjà avoir reçu un prototype de notre télécommande révolutionnaire pour votre examen. La télécommande comporte sept emplacements programmables (chacun pouvant être attribué à un appareil ménager différent) ainsi que des boutons marche/arrêt correspondants pour chacun. La télécommande dispose également d'un bouton d'annulation global.

Je joins également à cet e-mail un ensemble de classes Java qui ont été créées par divers fournisseurs pour contrôler les appareils domotiques tels que les lumières, les ventilateurs, les spas, les équipements audio et d'autres appareils contrôlables similaires.

Nous aimerais que vous créiez une API pour programmer la télécommande afin que chaque emplacement puisse être affecté au contrôle d'un appareil ou d'un ensemble d'appareils. Notez qu'il est important que nous puissions contrôler tous les appareils actuels, ainsi que tous les appareils futurs que les fournisseurs pourraient fournir.

Compte tenu du travail que vous avez effectué sur la station météo Weather-O-Rama, nous savons que vous ferez un excellent travail sur notre télécommande !

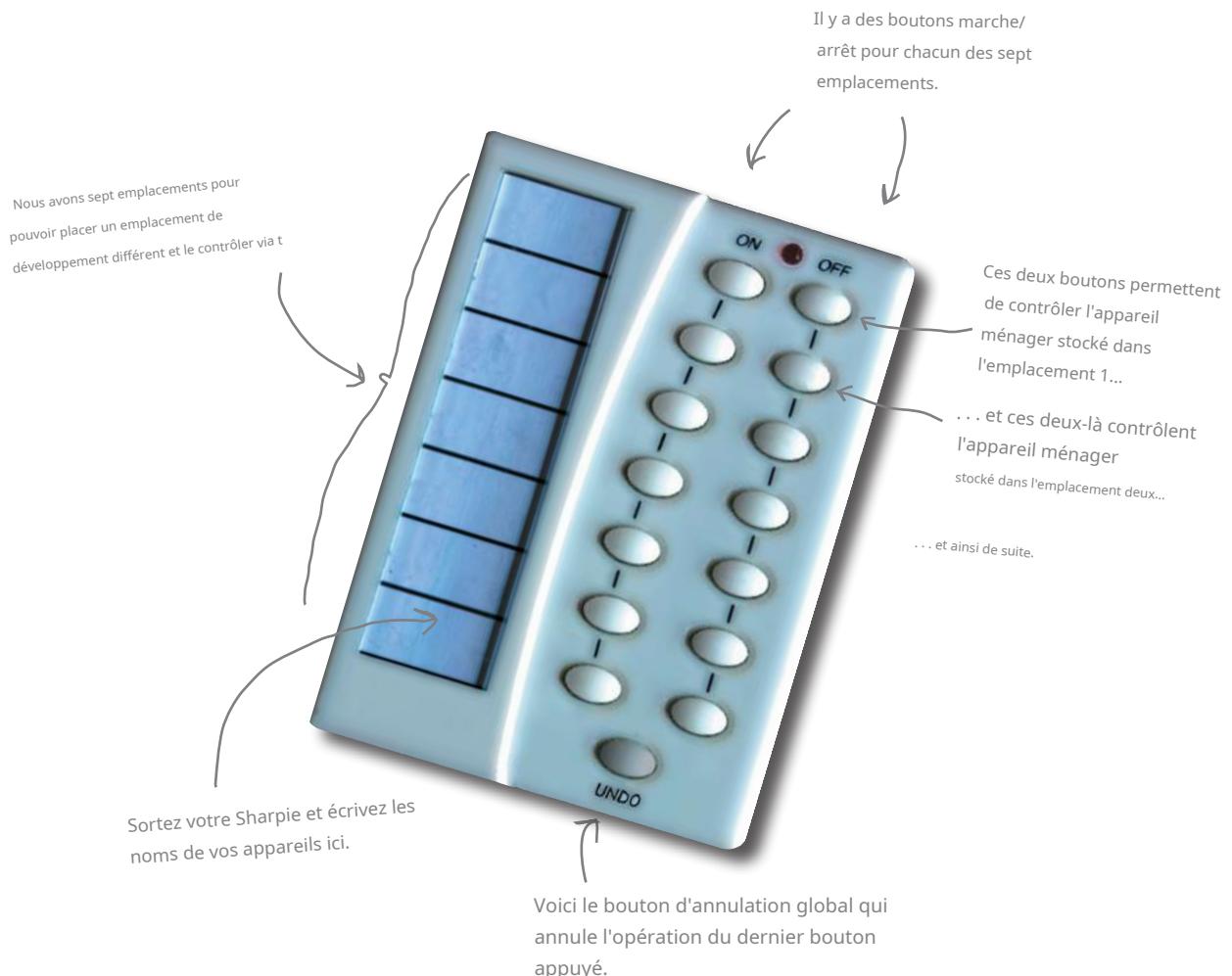
Nous avons hâte de voir votre conception.

Sincèrement,

Bill Thompson

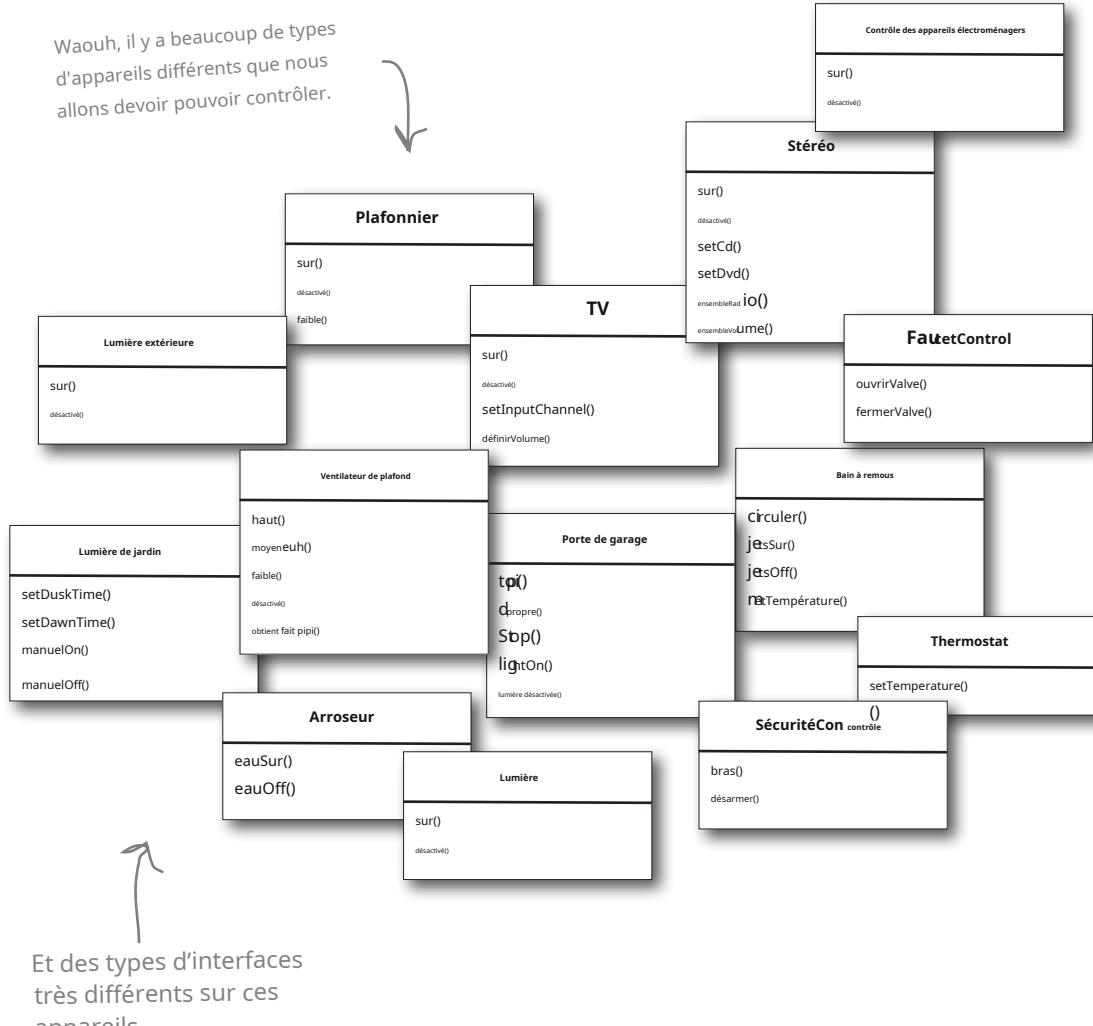
Bill Thompson, PDG

Du matériel gratuit ! Découvrons ensemble la télécommande...



Jetons un œil aux classes de fournisseurs

Examinons les classes de fournisseurs que le PDG a jointes à son e-mail.
Elles devraient vous donner une idée des interfaces des objets que nous devons contrôler à distance.



Il semble que nous ayons ici un ensemble assez conséquent de classes, et que l'industrie n'ait pas déployé beaucoup d'efforts pour proposer un ensemble d'interfaces communes. De plus, il semble que nous puissions nous attendre à voir apparaître davantage de ces classes à l'avenir. Concevoir une API de contrôle à distance va être intéressant. Passons à la conception.

Conversation en cabine

Vos coéquipiers discutent déjà de la manière de concevoir l'API de contrôle à distance...



Marie:Oui, je pensais que nous verrions un tas de classes avec les méthodes on() et off(), mais ici nous avons des méthodes comme dim(), setTemperature(), setVolume(), setInputChannel() et waterOn().

Poursuivre en justice:De plus, il semble que nous puissions nous attendre à davantage de classes de fournisseurs à l'avenir, avec des méthodes tout aussi diverses.

Marie:Je pense qu'il est important que nous considérons cela comme une séparation des préoccupations.

Poursuivre en justice:Signification?

Marie:Ce que je veux dire, c'est que la télécommande doit savoir interpréter les pressions sur les boutons et faire des demandes, mais elle ne doit pas en savoir beaucoup sur la domotique ou sur la façon d'allumer un spa.

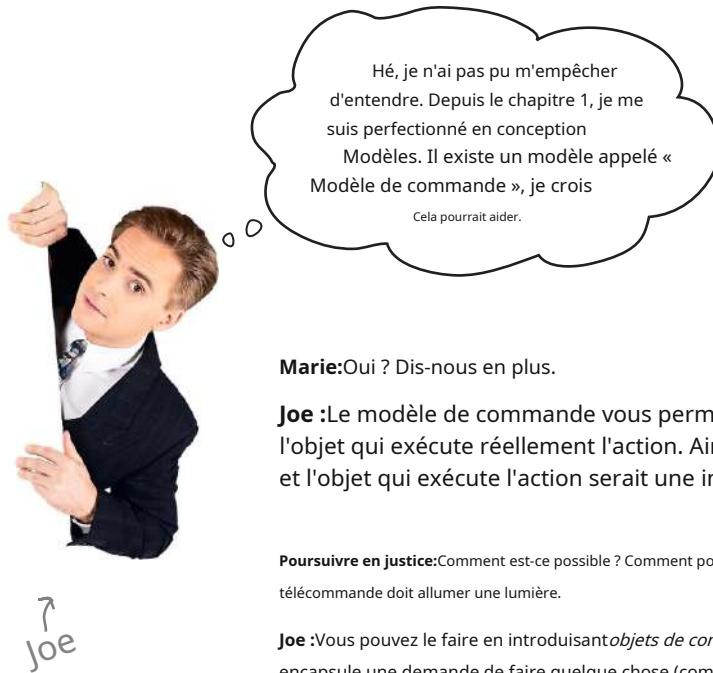
Poursuivre en justice:Mais si la télécommande est stupide et sait simplement faire des requêtes génériques, comment pouvons-nous concevoir la télécommande pour qu'elle puisse invoquer une action qui, par exemple, allume une lumière ou ouvre une porte de garage ?

Marie:Je ne suis pas sûr, mais nous ne voulons pas que la télécommande connaisse les spécificités des classes de fournisseurs.

Poursuivre en justice:Que veux-tu dire?

Marie:Nous ne voulons pas que la télécommande soit constituée d'un ensemble d'instructions if, comme « si slot1 == Light, then light.on(), else if slot1 == Hottub then hottub.jetsOn() ». Nous savons que c'est une mauvaise conception.

Poursuivre en justice:Je suis d'accord. Chaque fois qu'une nouvelle classe de fournisseur sort, nous devons intervenir et modifier le code, ce qui peut potentiellement créer des bugs et plus de travail pour nous-mêmes !



Marie:Oui ? Dis-nous en plus.

Joe :Le modèle de commande vous permet de découpler le demandeur d'une action de l'objet qui exécute réellement l'action. Ainsi, ici, le demandeur serait la télécommande et l'objet qui exécute l'action serait une instance de l'une de vos classes fournisseurs.

Poursuivre en justice:Comment est-ce possible ? Comment pouvons-nous les dissocier ? Après tout, lorsque j'appuie sur un bouton, la télécommande doit allumer une lumière.

Joe :Vous pouvez le faire en introduisant *objets de commandes* dans votre conception. Un objet de commande encapsule une demande de faire quelque chose (comme allumer une lumière) sur un objet spécifique (par exemple, l'objet d'éclairage du salon). Ainsi, si nous stockons un objet de commande pour chaque bouton, lorsque le bouton est enfoncé, nous demandons à l'objet de commande d'effectuer une tâche. La télécommande n'a aucune idée de la tâche à accomplir, elle dispose simplement d'un objet de commande qui sait comment communiquer avec le bon objet pour effectuer le travail. Vous voyez donc que la télécommande est découpée de l'objet d'éclairage !

Poursuivre en justice:Cela semble certainement aller dans la bonne direction.

Marie:Pourtant, j'ai du mal à comprendre ce modèle.

Joe :Étant donné que les objets sont si découplés, il est un peu difficile d'imaginer comment le modèle fonctionne réellement.

Marie:Voyons si j'ai au moins la bonne idée : en utilisant ce modèle, nous pourrions créer une API dans laquelle ces objets de commande peuvent être chargés dans des emplacements de bouton, ce qui permet au code de la télécommande de rester très simple. Et les objets de commande encapsulent la manière d'effectuer une tâche domotique ainsi que l'objet qui doit l'effectuer.

Joe :Oui, je pense. Je pense aussi que ce modèle peut vous aider avec ce bouton d'annulation, mais je n'ai pas encore étudié cette partie.

Marie:Cela semble vraiment encourageant, mais je pense qu'il me reste encore un peu de travail à faire pour vraiment « saisir » le modèle.

Poursuivre en justice:Moi aussi.

Pendant ce temps, de retour au Diner...,

OU,

Une brève introduction au modèle de commande

Comme Joe l'a dit, il est un peu difficile de comprendre le modèle de commande en entendant simplement sa description. Mais n'ayez pas peur, nous avons des amis prêts à vous aider : rappelez-vous

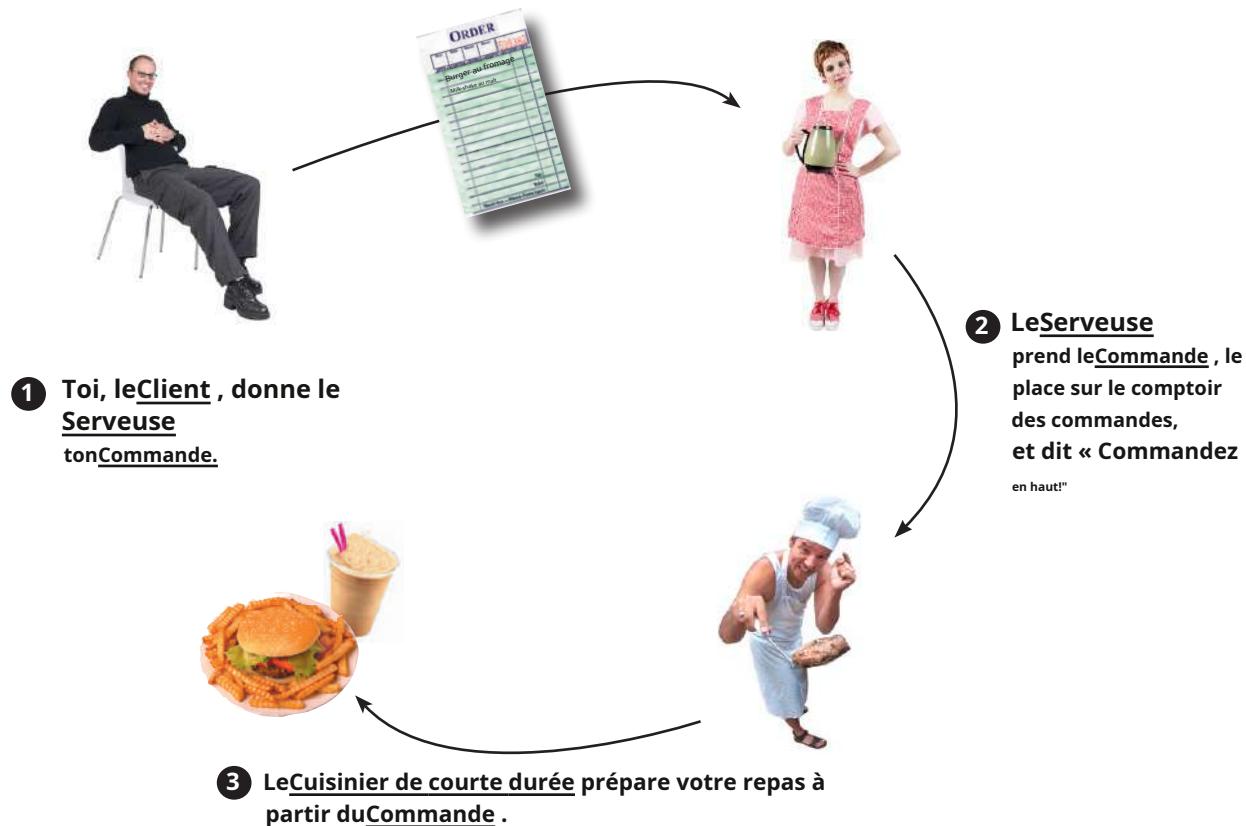
Notre sympathique restaurant du chapitre 1 ? Cela fait un moment que nous n'avons pas rendu visite à Alice, Flo et au cuisinier de la restauration rapide, mais nous avons une bonne raison de revenir (au-delà de la nourriture et de la bonne conversation) : le restaurant va nous aider à comprendre le modèle de commande.

Alors, faisons un petit détour vers le restaurant et étudions les interactions entre les clients, la serveuse, les commandes et le cuisinier. Grâce à ces interactions, vous allez comprendre les objets impliqués dans le modèle de commande et également avoir une idée du fonctionnement du découplage. Après cela, nous allons mettre au point cette API de contrôle à distance.



Enregistrement au Objectville Diner...

Ok, nous savons tous comment fonctionne le Diner :



Étudions l'interaction un peu plus en détail...

... et étant donné que ce Diner se trouve dans Objectville, réfléchissons également aux appels d'objets et de méthodes impliqués !



Rôles et responsabilités du Diner Objectville

Un bon de commande résume une demande de préparation d'un repas.

Considérez le bon de commande comme un objet qui agit comme une demande de préparation d'un repas. Comme tout objet, il peut être transmis de la serveuse au comptoir de commande ou à la serveuse suivante qui prend la relève. Il possède une interface composée d'une seule méthode, orderUp(), qui encapsule les actions nécessaires à la préparation du repas. Il possède également une référence à l'objet qui doit le préparer (dans notre cas, le cuisinier de la commande rapide). Il est encapsulé dans le sens où la serveuse n'a pas besoin de savoir ce que contient la commande ni même qui prépare le repas ; elle n'a qu'à passer le bon de commande dans la fenêtre de commande et à appeler « Order up ! »

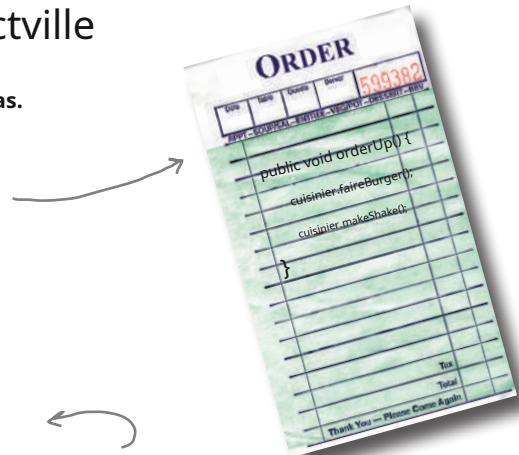
Le travail de la serveuse est de prendre les bons de commande et d'invoquer la méthode orderUp() sur eux.

La serveuse a la tâche facile : elle prend une commande du client, continue d'aider les clients jusqu'à ce qu'elle revienne au comptoir des commandes, puis invoque la méthode orderUp() pour préparer le repas. Comme nous l'avons déjà évoqué, dans Objectville, la serveuse ne se soucie pas vraiment de ce qui se trouve sur la commande ou de qui va la préparer ; elle sait juste que les bons de commande ont une méthode orderUp() qu'elle peut appeler pour faire le travail.

Désormais, tout au long de la journée, la méthode takeOrder() de la serveuse est paramétrée avec différents bons de commande de différents clients, mais cela ne la dérange pas ; elle sait que tous les bons de commande prennent en charge la méthode orderUp() et elle peut appeler orderUp() à chaque fois qu'elle a besoin qu'un repas soit préparé.

Le cuisinier de courte durée possède les connaissances nécessaires pour préparer le repas.

Le Short-Order Cook est l'objet qui sait vraiment comment préparer les repas. Une fois que la serveuse a invoqué la méthode orderUp(), le cuisinier prend le relais et implémente toutes les méthodes nécessaires à la création des repas. Notez que la serveuse et le cuisinier sont totalement découplés : la serveuse a des bons de commande qui encapsulent les détails du repas ; elle appelle simplement une méthode sur chaque commande pour la préparer. De même, le cuisinier obtient ses instructions à partir du bon de commande ; il n'a jamais besoin de communiquer directement avec la serveuse.



D'accord, dans la vraie vie, une serveuse se soucierait probablement de ce qui figure sur le bon de commande et de qui le cuisine, mais nous sommes à Objectville... travaillez avec nous ici !





Ok, nous avons un restaurant avec une serveuse qui est séparée du cuisinier de la restauration rapide par un bon de commande, et alors ? Allons droit au but !

Patience, on y arrive...

Considérez le Diner comme un modèle de conception OO qui nous permet de séparer un objet qui fait une demande des objets qui reçoivent et exécutent ces demandes. Par exemple, dans notre API de contrôle à distance, nous devons séparer le code qui est invoqué lorsque nous appuyons sur un bouton des objets des classes spécifiques au fournisseur qui exécutent ces demandes. Et si chaque emplacement de la télécommande contenait un objet comme l'objet Order Slip du Diner ? Ensuite, lorsqu'un bouton est enfoncé, nous pourrions simplement appeler l'équivalent de la méthode orderUp() sur cet objet et allumer les lumières sans que la télécommande ne sache comment faire en sorte que ces choses se produisent ou quels objets les font se produire.

Maintenant, changeons un peu de vitesse et mappons tout ce discours de Diner au modèle de commande...

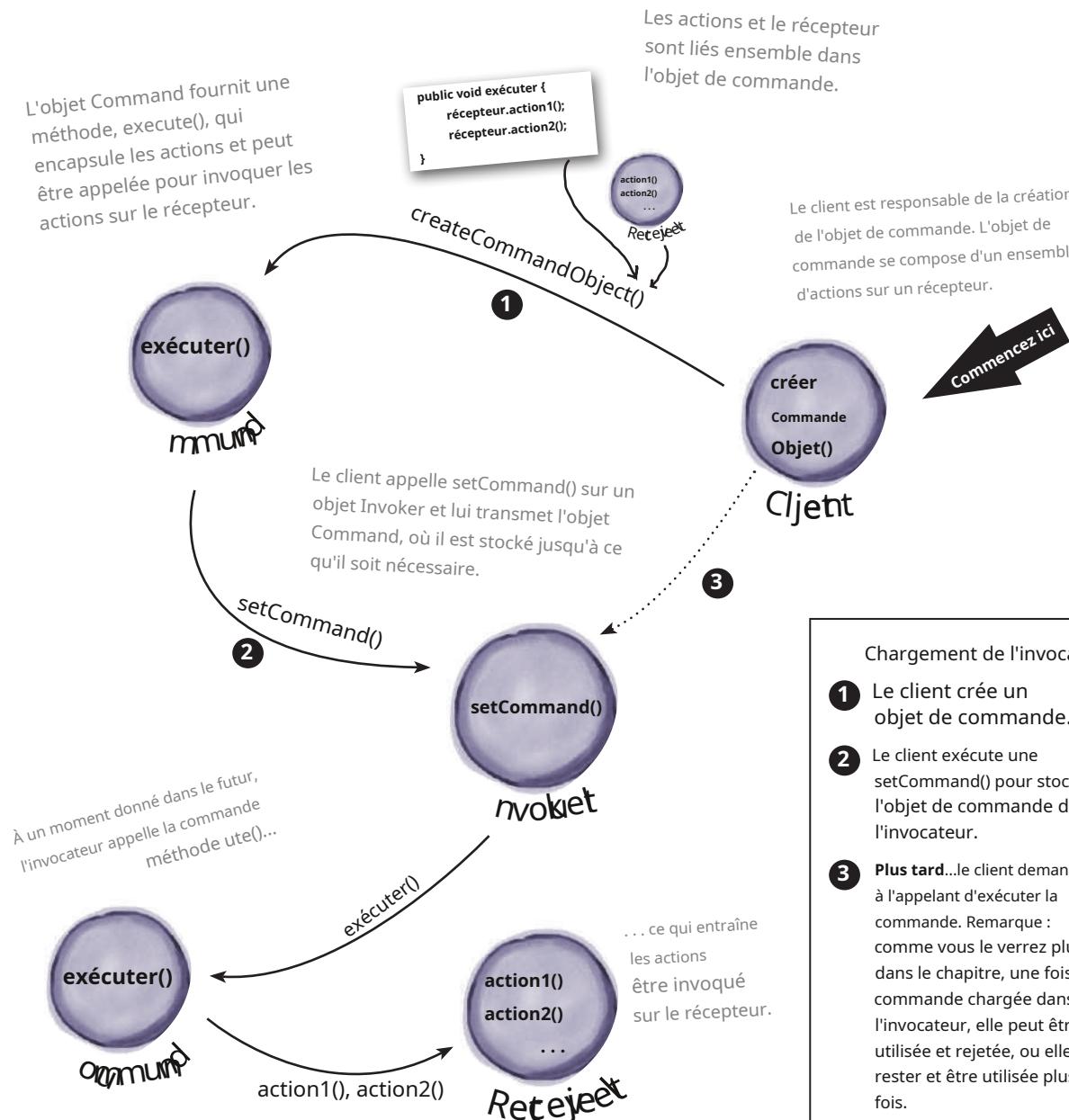


Avant de continuer, prenez le temps d'étudier le diagramme des deux pages précédentes, ainsi que les rôles et responsabilités des Diner, jusqu'à ce que vous pensiez avoir une idée des objets et des relations d'Objectville Diner. Une fois que vous avez fait cela, préparez-vous à maîtriser le modèle de commande !



Du Diner au Command Pattern

Bon, nous avons passé suffisamment de temps dans le Diner d'Objectville pour connaître assez bien toutes les personnalités et leurs responsabilités. Nous allons maintenant retravailler le diagramme du Diner pour refléter le modèle de commandement. Vous verrez que tous les joueurs sont les mêmes ; seuls les noms ont changé.



Chargement de l'invocateur

- 1 Le client crée un objet de commande.
- 2 Le client exécute une `setCommand()` pour stocker l'objet de commande dans l'invocateur.
- 3 Plus tard...le client demande à l'appelant d'exécuter la commande. Remarque : comme vous le verrez plus tard dans le chapitre, une fois la commande chargée dans l'invocateur, elle peut être utilisée et rejetée, ou elle peut rester et être utilisée plusieurs fois.



Faites correspondre les objets et les méthodes du restaurant avec les noms correspondants du modèle de commande.

Diner

Serveuse

Cuisinier de courte durée

commandeUp()

Commande

Client

prendreCommande()

Modèle de commande

Commande

exécuter()

Client

Invocateur

Récepteur

setCommand()

Notre premier objet de commande

N'est-il pas temps de construire notre premier objet de commande ? Allons-y et écrivons du code pour la télécommande. Bien que nous n'ayons pas encore compris comment concevoir l'API de la télécommande, construire quelques éléments de bas en haut peut nous aider...



Implémentation de l'interface de commande

Tout d'abord, tous les objets de commande implémentent la même interface, qui se compose d'une seule méthode. Dans Diner, nous avons appelé cette méthode `orderUp()`; cependant, nous utilisons généralement simplement le nom `execute()`.

Voici l'interface de commande :

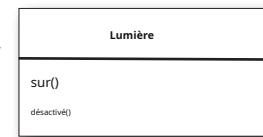
```
interface publique Commande {
    public void exécuter();
}
```



C'est simple. Tout ce dont nous avons besoin est une méthode appelée `execute()`.

Implémentation d'une commande pour allumer une lumière

Maintenant, supposons que vous souhaitez implémenter une commande pour allumer une lumière. En faisant référence à notre ensemble de classes de fournisseurs, la classe `Light` possède deux méthodes : `on()` et `off()`. Voici comment vous pouvez implémenter cela sous forme de commande :



Il s'agit d'une commande, nous devons donc implémenter l'interface de commande.

la classe publique `LightOnCommand` implémente `Command` {

Lumière lumière;



Le constructeur reçoit la lumière spécifique que cette commande va contrôler (par exemple la lumière du salon) et la stocke dans la variable d'instance `light`. Lorsque la méthode `execute()` est appelée, il s'agit de l'objet `light` qui va être le récepteur de la requête.

```
public LightOnCommand(Lumière lumière) {
    cette.lumière = lumière;
}
```

```
public void exécuter() {
    lumière.allumée();
}
```

La méthode `execute()` appelle la méthode `on()` sur l'objet récepteur, qui est la lumière que nous contrôlons.

Maintenant que vous avez une classe `LightOnCommand`, voyons si nous pouvons l'utiliser...

Utilisation de l'objet de commande

Bon, simplifions les choses : disons que nous avons une télécommande avec un seul bouton et un emplacement correspondant pour insérer un appareil à contrôler :

```
classe publique SimpleRemoteControl {  
    Emplacement de commande;  
    public SimpleRemoteControl() {}  
  
    public void setCommand(Commande commande) {  
        emplacement = commande;  
    }  
    public void buttonWasPressed() {  
        emplacement.execute();  
    }  
}
```

- Nous disposons d'un emplacement pour contenir notre commande, qui contrôlera un appareil.
- Nous avons une méthode pour définir la commande que l'emplacement va contrôler. Cela pourrait être appelé plusieurs fois si le client de ce code voulait changer le comportement du bouton de la télécommande.
- Cette méthode est appelée lorsque le bouton est enfoncé. Tout ce que nous faisons est de prendre la commande actuelle liée à l'emplacement et d'appeler sa méthode execute().

Créer un test simple pour utiliser la télécommande

Voici juste un peu de code pour tester la télécommande simple. Jetons un œil et nous vous montrerons comment les pièces correspondent au diagramme de modèle de commande :

```
classe publique RemoteControlTest {  
    public static void main(String[] args) {  
        SimpleRemoteControl remote = new SimpleRemoteControl();  
        Lumière lumière = new Lumière();  
        LightOnCommand lightOn = new LightOnCommand(lumière);  
  
        remote.setCommand(lumière  
        allumée); remote.buttonWasPressed();  
    }  
}
```

Et puis nous simulons le bouton enfoncé.

- Il s'agit de notre modèle de client en langage de commande.
- La télécommande est notre Invoker ; un objet de commande lui sera transmis qui pourra être utilisé pour effectuer des requêtes.
- Nous créons maintenant un objet Light. Il sera le récepteur de la requête.

Ici, créez une commande et transmettez-lui le récepteur.

Voici le résultat de l'exécution de ce code de test:

Fenêtre d'aide pour l'édition de fichiers DinerFoodYum

% java RemoteControlTest

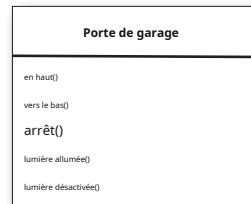
La lumière est allumée

%



Ok, il est temps pour vous d'implémenter la classe GarageDoorOpenCommand.
Tout d'abord, fournissez le code de la classe ci-dessous. Vous aurez besoin du diagramme de classe GarageDoor.

classe publique GarageDoorOpenCommand
implémente la commande {



Votre code ici

}

Maintenant que vous avez votre classe, quel est le résultat du code suivant ? (Astuce : la méthode GarageDoor up() affiche « La porte du garage est ouverte » lorsqu'elle est terminée.)

```

classe publique RemoteControlTest {
    public static void main(String[] args) {
        SimpleRemoteControl remote = new SimpleRemoteControl();
        Lumière lumière = new Lumière();
        GarageDoor garageDoor = new GarageDoor();
        CommandeLumièreActivée lightOn = new LightOnCommand(lumière);
        CommandePorteGarageOuverture garageOpen =
            nouvelle GarageDoorOpenCommand(garageDoor);

        remote.setCommand(lumière allumée);
        remote.buttonWasPressed();
        remote.setCommand(garageOpen);
        remote.buttonWasPressed();
    }
}

```

Votre sortie ici.



Le modèle de commande défini

Vous avez terminé votre travail dans Objectville Diner, vous avez partiellement implémenté l'API de contrôle à distance et, ce faisant, vous avez une assez bonne idée de la manière dont les classes et les objets interagissent dans le modèle de commande. Nous allons maintenant définir le modèle de commande et en préciser tous les détails.

Commençons par sa définition officielle :

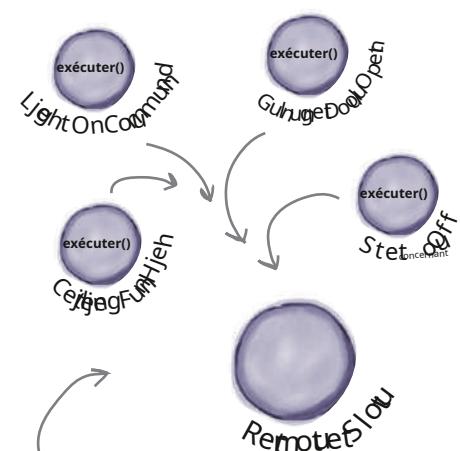
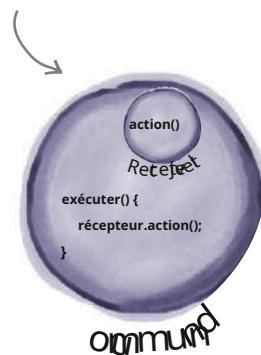
Le modèle de commande encapsule une requête en tant qu'objet, vous permettant ainsi de paramétriser d'autres objets avec des requêtes différentes, de mettre en file d'attente ou de consigner des requêtes et de prendre en charge les opérations annulables.

Voyons cela en détail. Nous savons qu'un objet de commande *encapsule une requête* en liant ensemble un ensemble d'actions sur un récepteur spécifique. Pour y parvenir, il regroupe les actions et le récepteur dans un objet qui expose une seule méthode, `execute()`. Lorsqu'elle est appelée, `execute()` provoque l'invocation des actions sur le récepteur. De l'extérieur, aucun autre objet ne sait vraiment quelles actions sont exécutées sur quel récepteur ; ils savent juste que s'ils appellent la méthode `execute()`, leur requête sera traitée.

Nous avons également vu quelques exemples de *paramétriser un objet* avec une commande. De retour au restaurant, la serveuse a été paramétrée avec plusieurs commandes tout au long de la journée. Dans la télécommande simple, nous avons d'abord chargé l'emplacement du bouton avec une commande « allumer la lumière », puis l'avons remplacé plus tard par une commande « ouvrir la porte du garage ». Comme la serveuse, votre emplacement à distance ne se souciait pas de l'objet de commande qu'il avait, tant qu'il implementait l'interface de commande.

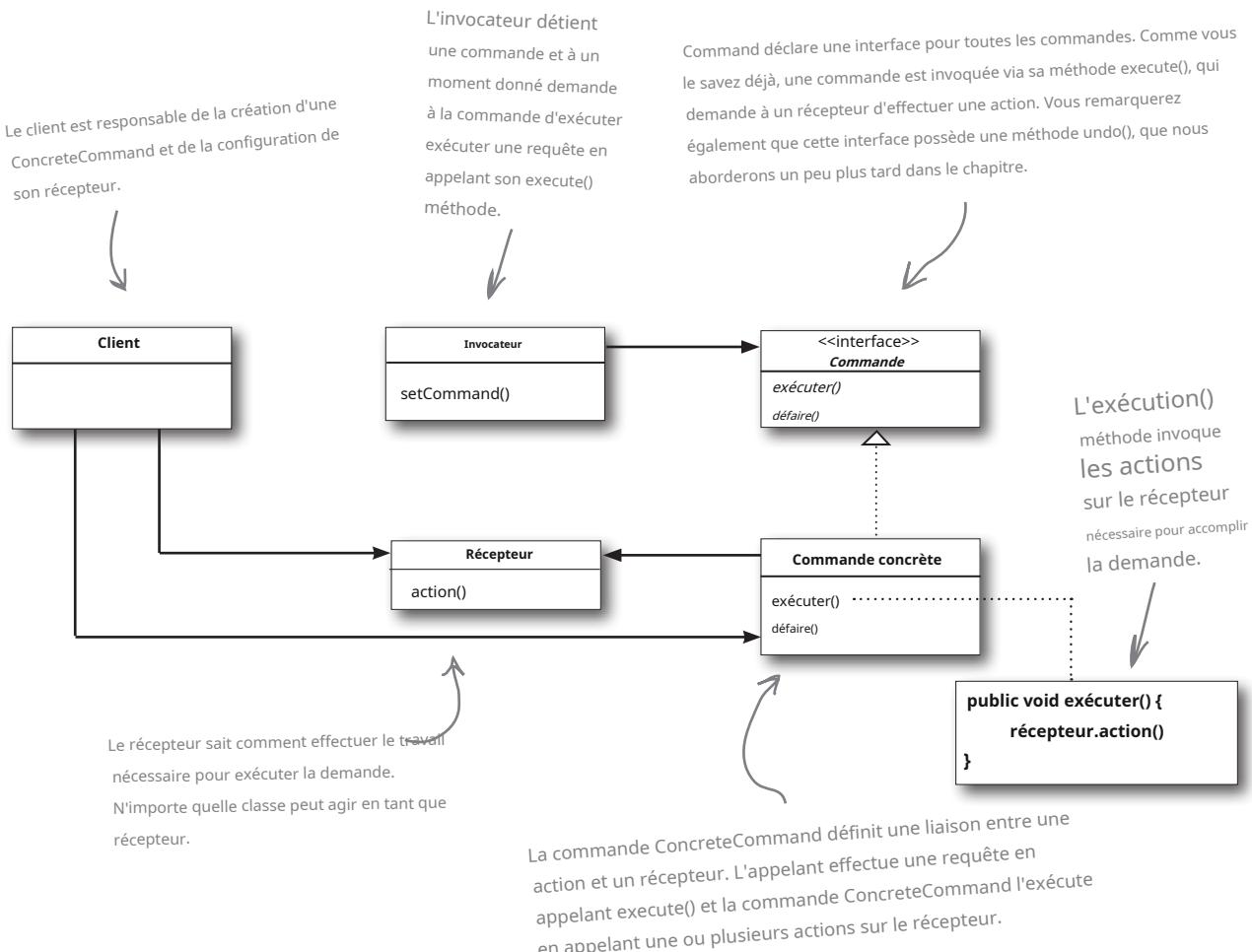
Ce que nous n'avons pas encore rencontré, c'est l'utilisation de commandes pour implémenter *files d'attente et journaux et prise en charge des opérations d'annulation*. Ne vous inquiétez pas, ce sont des extensions assez simples du modèle de commande de base, et nous y reviendrons bientôt. Nous pouvons également facilement prendre en charge ce que l'on appelle le modèle de commande métà une fois que nous avons mis en place les bases. Le modèle de commande métà vous permet de créer des macros de commandes afin de pouvoir exécuter plusieurs commandes à la fois.

Une demande encapsulée.

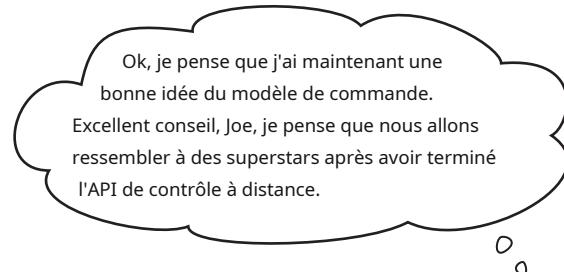


Un invocateur — par exemple, un emplacement de la télécommande — peut être paramétré avec différentes requêtes.

Le modèle de commande défini : le diagramme de classes



Comment la conception du modèle de commande prend-elle en charge le découplage de l'appelant d'une requête et du récepteur de la requête ?



Marie: Moi aussi. Alors par où commencer ?

Poursuivre en justice: Comme nous l'avons fait dans SimpleRemote, nous devons fournir un moyen d'assigner des commandes aux emplacements. Dans notre cas, nous avons sept emplacements, chacun avec un bouton marche/arrêt. Nous pourrions donc assigner des commandes à la télécommande comme ceci :

onCommands[0] = onCommand;
offCommands[0] = offCommand;

et ainsi de suite pour chacun des sept emplacements de commande.

Marie: Cela a du sens, sauf pour les objets Light. Comment la télécommande fait-elle la différence entre la lumière du salon et celle de la cuisine ?

Poursuivre en justice: Ah, c'est exactement ça, ce n'est pas le cas ! La télécommande ne sait rien d'autre que comment appeler execute() sur l'objet de commande correspondant lorsqu'un bouton est enfoncé.

Marie: Ouais, j'ai en quelque sorte compris, mais dans l'implémentation, comment pouvons-nous nous assurer que les bons objets allument et éteignent les bons appareils ?

Poursuivre en justice: Lorsque nous créons les commandes à charger dans la télécommande, nous créons une LightCommand liée à l'objet d'éclairage du salon et une autre liée à l'objet d'éclairage de la cuisine. N'oubliez pas que le récepteur de la requête est lié à la commande dans laquelle elle est encapsulée. Ainsi, au moment où le bouton est enfoncé, personne ne se soucie de savoir quelle lumière est laquelle ; la bonne chose se produit simplement lorsque la méthode execute() est appelée.

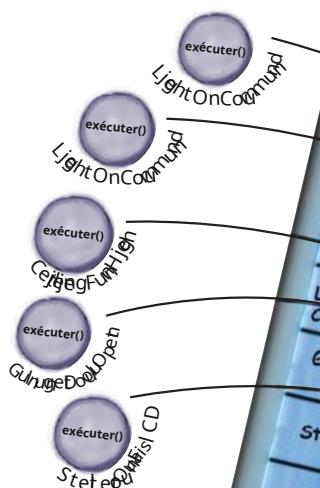
Marie: Je pense que j'ai compris. Implémentons la télécommande et je pense que cela deviendra plus clair !

Poursuivre en justice: Ça a l'air bien. Essayons-le...

Affectation de commandes aux emplacements

Nous avons donc un plan : nous allons attribuer une commande à chaque emplacement de la télécommande. Cela fait de la télécommande notre *invocateur*. Lorsqu'un bouton est enfoncé, la méthode *execute()* sera appelée sur la commande correspondante, ce qui entraînera l'appel d'actions sur le récepteur (comme des lumières, des ventilateurs de plafond et des chaînes stéréo).

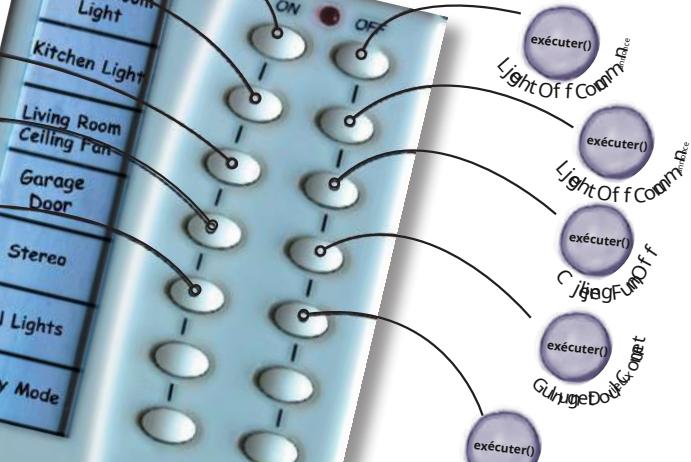
(1) Chaque emplacement reçoit une commande.



Nous nous occuperons des emplacements restants dans un instant.

Dans notre code, vous constaterez que chaque nom de commande est suivi du mot « Commande », mais dans la version imprimée, nous n'avons malheureusement plus de place pour quelques-uns d'entre eux.

(2) Lorsque le bouton est enfoncé, la méthode *execute()* est appelée sur la commande correspondante.



(3) Dans la méthode *execute()*, on est actes invoqué sur le récepteur.



Mise en œuvre de la télécommande

```
classe publique RemoteControl {
```

```
    Commande[] onCommands;
```

```
    Commande[] offCommands;
```

Cette fois-ci, la télécommande va gérer sept commandes marche/arrêt, que nous conserverons dans des tableaux correspondants.

```
public RemoteControl() {
```

```
    onCommands = nouvelle commande[7];
```

```
    offCommands = nouvelle commande[7];
```

Dans le constructeur, tout ce que nous devons faire est d'instancier et d'initialiser les tableaux On et Off.

```
    Commande noCommand = new
```

```
    NoCommand(); pour (int i = 0; i < 7; i++) {
```

```
        onCommands[i] = noCommand;
```

```
        offCommands[i] = noCommand;
```

```
}
```

```
}
```

La méthode setCommand() prend une position d'emplacement et une commande On et Off à stocker dans cet emplacement.

```
public void setCommand(int slot, Commande onCommand, Commande offCommand) {
```

```
    onCommands[slot] = onCommand;
```

```
    offCommands[slot] = offCommand;
```

```
}
```

Il place ces commandes dans les tableaux On et Off pour une utilisation ultérieure.

```
public void onButtonWasPushed(int slot) {
```

```
    onCommands[emplacement].execute();
```

```
}
```

Lorsqu'un bouton On ou Off est enfoncé, le matériel se charge d'appeler les méthodes correspondantes onButtonWasPushed() ou offButtonWasPushed().

```
public void offButtonWasPushed(int slot) {
```

```
    offCommands[emplacement].execute();
```

```
}
```

```
chaîne publique toString() {
```

```
    StringBuffer stringBuff = new StringBuffer();
```

```
    stringBuff.append("\n----- Télécommande ----- \n"); for (int i = 0; i <
```

```
    onCommands.length; i++) {
```

```
        stringBuff.append("[emplacement " + i + "] " + onCommands[i].getClass().getName()
            + " " + offCommands[i].getClass().getName() + "\n");
```

```
}
```

```
renvoie stringBuff.toString();
```

```
}
```

Nous remplaçons toString() pour imprimer chaque emplacement et sa commande correspondante. Vous nous verrez l'utiliser lorsque nous testerons la télécommande.

210

Chapitre 6

Mise en œuvre des commandes

Eh bien, nous avons déjà mis en œuvre la commande LightOnCommand pour la SimpleRemoteControl. Nous pouvons insérer le même code ici et tout fonctionne à merveille. Les commandes Off ne sont pas différentes ; en fait, la commande LightOffCommand ressemble à ceci :

la classe publique LightOffCommand implémente Command {

Lumière lumière;

```
public LightOffCommand(Lumière lumière) {
```

```
    cette.lumière = lumière;
```

```
}
```

```
public void exécuter() {
```

```
    lumière.éteinte();
```

```
}
```

```
}
```

La LightOffCommand fonctionne exactement de la même manière que la LightOnCommand, sauf que nous lisons le récepteur à une action différente : la méthode off().

Essayons quelque chose d'un peu plus difficile ; pourquoi ne pas écrire des commandes on et off pour la stéréo ? Ok, off est facile, nous lisons simplement la stéréo à la méthode off() dans la StereoOffCommand. On est un peu plus compliqué ; disons que nous voulons écrire une StereoOnWithCDCommand...

Stéréo

sur()
désactivé()
setCd()
setDvd()
setRadio()
définirVolume()

classe publique StereoOnWithCDCommand implémente Command {

Stéréo stéréo;

```
public StereoOnWithCDCommand(Stereo stéréo) {
```

```
    ceci.stereo = stéréo;
```

```
}
```

Tout comme LightOnCommand, nous recevons l'instance de la chaîne stéréo que nous allons contrôler et nous la stockons dans une variable d'instance.

```
public void exécuter() {
```

```
    stéréo.on();
```

```
    stéréo.setCd();
```

```
    stéréo.setVolume(11);
```

```
}
```

```
}
```

Pour exécuter cette requête, nous devons appeler trois méthodes sur la chaîne stéréo : d'abord, l'allumer, puis la régler pour lire le CD, et enfin régler le volume sur 11. Pourquoi 11 ? Eh bien, c'est mieux que 10, non ?

Pas trop mal. Jetez un œil au reste des classes de fournisseurs ; à présent, vous pouvez certainement éliminer le reste des classes de commande dont nous avons besoin pour celles-ci.

Mettre la télécommande à l'épreuve

Notre travail avec la télécommande est pratiquement terminé ; il ne nous reste plus qu'à exécuter quelques tests et rassembler de la documentation pour décrire l'API. Home Automation or Bust, Inc. va être impressionné, vous ne pensez pas ? Nous avons réussi à mettre au point une conception qui leur permettra de produire une télécommande facile à entretenir, et ils n'auront aucun mal à convaincre les fournisseurs d'écrire des classes de commandes simples à l'avenir, car elles sont très faciles à écrire.

Passons au test de ce code !

classe publique RemoteLoader {

```
public static void main(String[] args) {  
    Télécommande remoteControl = new RemoteControl();
```

```
Lumière livingRoomLight = new Light("Salon"); Lumière  
kitchenLight = new Light("Cuisine"); Ventilateur de plafond  
ceilingFan = new CeilingFan("Salon"); Porte de garage Porte de  
garage = new GarageDoor("Garage"); Stéréo stéréo = new  
Stereo("Salon");
```

Créez tous les appareils à leurs emplacements appropriés.

```
Commande LightOn livingRoomOn =  
    nouvelle commande LightOn(livingRoomLight);  
commande LightOff livingRoomOff =  
    nouvelle commande LightOff(livingRoomLight);  
commande LightOn kitchenLightOn =  
    nouvelle commande LightOn(kitchenLight);  
commande LightOff kitchenLightOff =  
    nouvelle commande LightOffCommand(kitchenLight);
```

Créez tous les objets de commande Light.

```
Commande CeilingFanOn ceilingFanOn =  
    nouvelle commande CeilingFanOnCommand(ceilingFan);  
commande CeilingFanOff ceilingFanOff =  
    nouvelle commande CeilingFanOffCommand(ceilingFan);
```

Créez le bouton Marche/Arrêt pour le ventilateur de plafond.

```
Commande GarageDoorUp garageDoorUp =  
    nouvelle commande GarageDoorUp(garageDoor);  
commande GarageDoorDown garageDoorDown =  
    nouvelle commande GarageDoorDown(garageDoor);
```

Créez les commandes Haut et Bas pour le Garage.

```
Commande StereoOnWithCD stereoOnWithCD =  
    nouvelle StereoOnWithCDCommand(stéréo);  
StereoOffCommand stereoOff =  
    nouvelle StereoOffCommand(stéréo);
```

Créez les commandes marche/arrêt stéréo.

```

remoteControl.setCommand(0, éclairage du salon allumé, éclairage du salon éteint);
remoteControl.setCommand(1, éclairage de la cuisine allumé, éclairage de la cuisine éteint);
remoteControl.setCommand(2, ventilateur de plafond allumé, ventilateur de plafond éteint);
remoteControl.setCommand(3, stéréo allumée avec CD, stéréo éteinte);

Système.out.println(télécommande);

remoteControl.onButtonWasPushed(0);
remoteControl.offButtonWasPushed(0);
remoteControl.onButtonWasPushed(1);
remoteControl.offButtonWasPushed(1);
remoteControl.onButtonWasPushed(2);
remoteControl.offButtonWasPushed(2);
remoteControl.onButtonWasPushed(3);
remoteControl.offButtonWasPushed(3);
}

}

```

Maintenant que nous avons toutes nos commandes, nous pouvons les charger dans les emplacements distants.

C'est ici que nous utilisons notre méthode `toString()` pour imprimer chaque emplacement distant et la commande qui lui est attribuée. (Notez que `toString()` est appelé automatiquement ici, nous n'avons donc pas besoin d'appeler `toString()` explicitement.)

Très bien, nous sommes prêts à démarrer ! Maintenant, nous parcourons chaque emplacement et appuyons sur ses boutons Marche et Arrêt.

Voyons maintenant l'exécution de notre test de télécommande...

```

Aide de la fenêtre d'édition de fichier CommandesGetThingsDone

% Chargeur à distance Java
----- Télécommande -----
[emplacement 0] LightOnCommand
[emplacement 1] Commande d'éclairage allumé [emplacement 2]
Commande de ventilateur de plafond allumé [emplacement 3]
Commande stéréo allumée avec CD [emplacement 4] Aucune
commande
[emplacement 5] NoCommand
[emplacement 6] NoCommand
Sur les machines à sous Hors machines à sous
La lumière du salon est allumée La
lumière du salon est éteinte La
lumière de la cuisine est allumée
La lumière de la cuisine est éteinte
Le ventilateur de plafond du salon est allumé à fond.
Le ventilateur de plafond du salon est éteint. La chaîne
stéréo du salon est allumée.
La chaîne stéréo du salon est réglée sur l'entrée CD Le
volume de la chaîne stéréo du salon est réglé sur 11 La
chaîne stéréo du salon est éteinte
%

```

Nos commandes en action ! N'oubliez pas que la sortie de chaque périphérique provient des classes du fournisseur. Par exemple, lorsqu'un objet lumineux est allumé, il affiche « L'éclairage du salon est allumé ».



Bonne prise. Nous avons réussi à glisser un petit quelque chose là-dedans. Dans la télécommande, nous ne voulions pas vérifier si une commande était chargée à chaque fois que nous référencions un emplacement. Par exemple, dans la méthode onButtonWasPushed(), nous aurions besoin d'un code comme celui-ci :

```
public void onButtonWasPushed(int slot) {
    si (onCommands[slot] != null) {
        onCommands[emplacement].execute();
    }
}
```

Alors, comment contourner ce problème ? Implémentez une commande qui ne fait rien !

```
la classe publique NoCommand implémente Command {
    public void exécuter() { }
}
```

Ensuite, dans notre constructeur RemoteControl, nous attribuons à chaque emplacement un objet NoCommand par défaut et nous savons que nous aurons toujours une commande à appeler dans chaque emplacement.

```
Commande noCommand = new
NoCommand(); pour (int i = 0; i < 7; i++) {
    onCommands[i] = noCommand;
    offCommands[i] = noCommand;
}
```

Ainsi, dans la sortie de notre test, vous ne voyez que les emplacements qui ont été attribués à une commande autre que l'objet NoCommand par défaut, que nous avons attribué lors de la création du constructeur RemoteControl.



Modèle Honorable Mention

L'objet NoCommand est un exemple de *objet nul*. Un objet nul est utile lorsque vous n'avez pas d'objet significatif à renvoyer, et que vous souhaitez néanmoins supprimer la responsabilité de la gestion **nul** du client. Par exemple, dans notre télécommande, nous n'avions pas d'objet significatif à attribuer à chaque emplacement par défaut, nous avons donc fourni un objet NoCommand qui agit comme un substitut et ne fait rien lorsque sa méthode execute() est appelée.

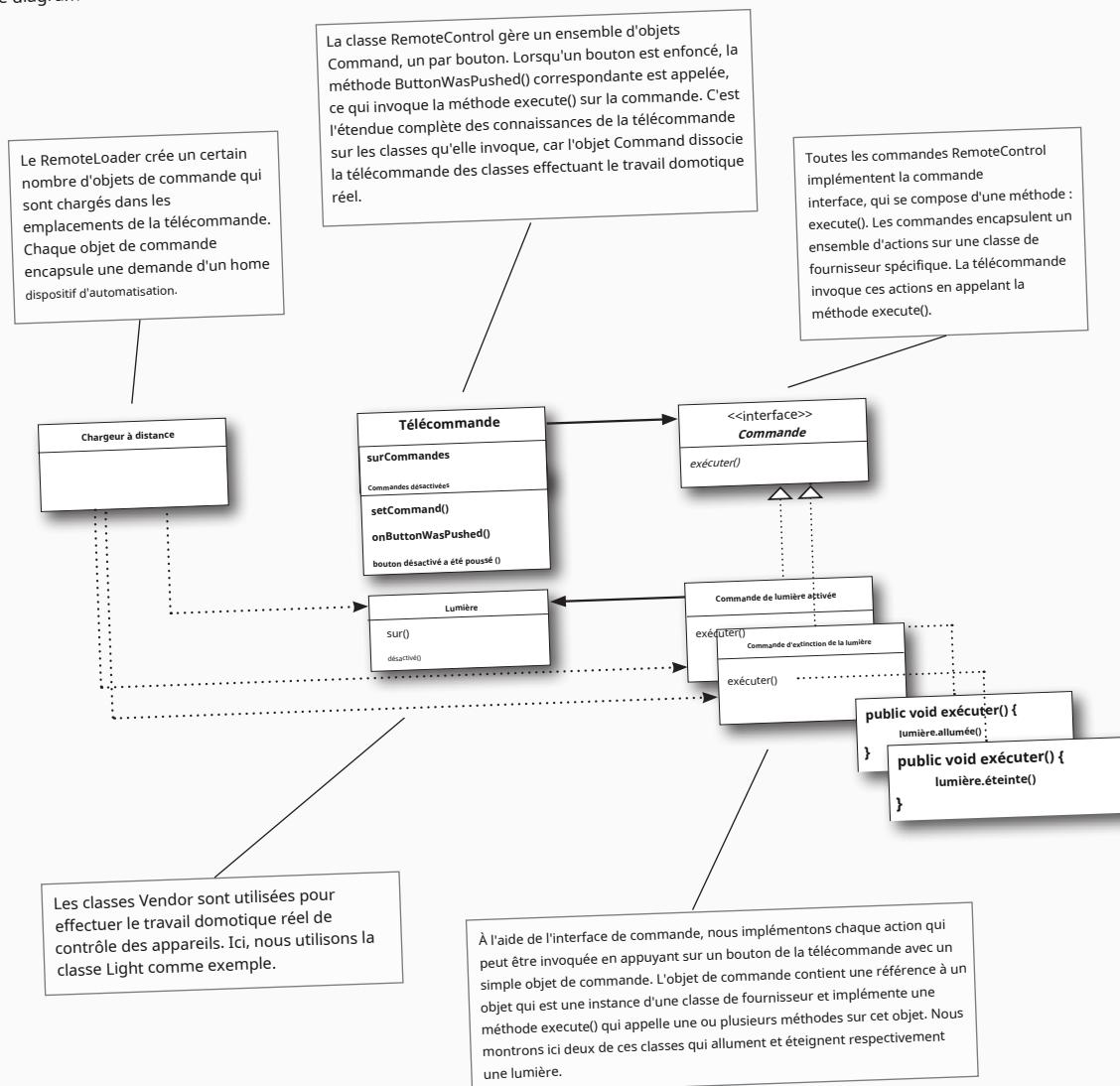
Vous trouverez des utilisations pour les objets nuls en conjonction avec de nombreux modèles de conception, et parfois vous verrez même « Objet nul » répertorié comme modèle de conception.

Il est temps d'écrire cette documentation...

Conception d'API de contrôle à distance pour la domotique ou Bust, Inc.

Nous sommes heureux de vous présenter l'interface de conception et de programmation d'application suivante pour votre télécommande domotique. Notre objectif de conception principal était de garder le code de la télécommande aussi simple que possible afin qu'il ne nécessite pas de modifications à mesure que de nouvelles classes de fournisseurs sont produites. À cette fin, nous avons utilisé le modèle de commande pour découpler logiquement la classe RemoteControl des classes de fournisseurs. Nous pensons que cela réduira le coût de production de la télécommande ainsi que vos coûts de maintenance en cours.

Le diagramme de classe suivant donne un aperçu de notre conception :





Serious Coding

Vous souhaitez faire passer votre codage de modèles de commande au niveau supérieur ? Vous pouvez utiliser les expressions lambda de Java pour ignorer l'étape de création de tous ces objets de commande concrets. Avec les expressions lambda, au lieu d'instancier les objets de commande concrets, vous pouvez utiliser *objets de fonction* à leur place. En d'autres termes, nous pouvons utiliser un objet fonction *comme une commande* et, pendant que nous y sommes, nous pouvons également supprimer toutes ces classes de commandes concrètes.

Voyons comment vous utiliserez les expressions lambda comme commandes pour simplifier notre code précédent :

Le code mis à jour, utilisant des expressions lambda :

```
classe publique RemoteLoader {
```

```
    public static void main(String[] args) {  
        Télécommande remoteControl = new RemoteControl();
```

Nous créons l'objet Lumière
comme d'habitude...

```
        Lumière livingRoomLight = new Light("Salon"); ...
```

Mais nous pouvons
enlever le béton
LightOnCommand et
Commande d'extinction de la lumière
objets.

```
        Commande LightOn livingRoomLightOn =—  
                nouvelle LightOnCommand(livingRoomLight);
```

```
        Commande LightOff livingRoomLightOff =—  
                nouvelle commande LightOffCommand(livingRoomLight);
```

```
        ...  
        remoteControl.setCommand(0, () -> livingRoomLight.on(),  
                                () -> livingRoomLight.off());
```

Au lieu de cela, nous écrirons les commandes concrètes sous
forme d'expressions lambda qui font le même travail que la
méthode execute() de la commande concrète : c'est-à-dire
allumer la lumière ou l'éteindre.

```
    }  
    ...
```

Plus tard, lorsque vous cliquez sur l'un des boutons de
la télécommande, la télécommande appelle la
méthode execute() de l'objet de commande dans
l'emplacement de ce bouton, qui est représenté par
cette expression lambda.

Une fois que nous avons remplacé les commandes concrètes par des expressions lambda, nous pouvons supprimer toutes
ces classes de commandes concrètes (LightOnCommand, LightOffCommand, HottubOnCommand, HottubOffCommand,
etc.). Si vous faites cela pour chaque commande concrète, vous réduirez le nombre total de classes dans l'application de
contrôle à distance de 22 à 9.

Notez que vous ne pouvez le faire que si votre interface de commande possède *une* méthode abstraite. Dès que nous ajoutons
une deuxième méthode abstraite, le raccourci lambda ne fonctionne plus.

Si vous aimez cette technique, consultez votre référence Java préférée pour plus d'informations sur l'expression
lambda.



Oups ! Nous avons presque oublié... Heureusement, une fois que nous avons nos classes de commandes de base, l'annulation est facile à ajouter. Passons en revue l'ajout d'une annulation à nos commandes et à la télécommande...

Que faisons-nous ?

Ok, nous devons ajouter une fonctionnalité pour prendre en charge le bouton Annuler sur la télécommande. Cela fonctionne comme ceci : disons que la lumière du salon est éteinte et que vous appuyez sur le bouton Marche de la télécommande. Évidemment, la lumière s'allume. Maintenant, si vous appuyez sur le bouton Annuler, la dernière action sera inversée — dans ce cas, la lumière s'éteindra. Avant d'aborder des exemples plus complexes, faisons fonctionner la lumière avec le bouton Annuler :

- 1 Lorsque les commandes prennent en charge l'annulation, elles disposent d'une méthode undo() qui reflète la méthode execute(). Quel que soit le dernier résultat obtenu par execute(), undo() l'inverse. Ainsi, avant de pouvoir ajouter l'annulation à nos commandes, nous devons ajouter une méthode undo() à l'interface Command :

```
interface publique Commande {
    public void exécuter();
    public void annuler();
}
```



Voici la nouvelle méthode undo().

C'était assez simple.

Maintenant, plongeons dans les commandes Light et implémentons la méthode undo().

- 2 Commençons par LightOnCommand : si la méthode execute() de LightOnCommand a été appelée, alors la méthode on() a été appelée en dernier. Nous savons que undo() doit faire le contraire en appelant la méthode off().

```
la classe publique LightOnCommand implémente Command {  
    Lumière lumière;  
  
    public LightOnCommand(Lumière lumière) {  
        cette.lumière = lumière;  
    }  
  
    public void exécuter() {  
        lumière.allumée();  
    }  
  
    public void annuler() {  
        lumière.éteinte();  
    }  
}
```

Un jeu d'enfant ! Passons maintenant à la commande LightOffCommand. Ici, la méthode undo() doit simplement appeler la méthode on() de la lumière.

```
la classe publique LightOffCommand implémente Command {  
    Lumière lumière;  
  
    public LightOffCommand(Lumière lumière) {  
        cette.lumière = lumière;  
    }  
  
    public void exécuter() {  
        lumière.éteinte();  
    }  
  
    public void annuler() {  
        lumière.allumée();  
    }  
}
```

← Et ici, undo() rallume la lumière.

Cela pourrait-il être plus simple ? Bon, nous n'avons pas encore terminé ; nous devons intégrer un peu de support dans la télécommande pour gérer le suivi du dernier bouton appuyé et du bouton d'annulation.

3

Pour ajouter la prise en charge du bouton Annuler, il suffit d'apporter quelques petites modifications à la classe Remote Control. Voici comment nous allons procéder : nous ajouterons une nouvelle variable d'instance pour suivre la dernière commande invoquée ; ensuite, chaque fois que le bouton Annuler est enfoncé, nous récupérons cette commande et invoquons sa méthode undo().

classe publique RemoteControlWithUndo {

```
Commande[] onCommands;
Commande[] offCommands;
Commande undoCommand;
```

C'est ici que nous allons stocker la dernière commande exécutée pour le bouton Annuler.

```
public RemoteControlWithUndo() {
    onCommands = nouvelle commande[7];
    offCommands = nouvelle commande[7];
```

```
Commande noCommand = new
NoCommand(); pour(int i=0;i<7;i++) {
    onCommands[i] = noCommand;
    offCommands[i] = noCommand;
}
```

Tout comme les autres emplacements, l'annulation commence par une noCommand, donc appuyer sur Annuler avant n'importe quel autre bouton ne fera rien du tout.

```
annulerCommande = aucuneCommande;
```

```
}
```

```
public void setCommand(int slot, Commande onCommand, Commande offCommand) {
    onCommands[slot] = onCommand;
    offCommands[slot] = offCommand;
}
```

```
public void onButtonWasPushed(int slot) {
    onCommands[emplacement].execute();
    undoCommand = onCommands[emplacement];
}
```

Lorsqu'un bouton est enfoncé, nous prenons la commande et l'exécutons d'abord, puis nous enregistrons une référence à celle-ci dans la variable d'instance undoCommand. Nous faisons cela pour les commandes on et off.

```
public void offButtonWasPushed(int slot) {
    offCommands[emplacement].execute();
    undoCommand = offCommands[emplacement];
}
```

```
public void undoButtonWasPushed() {
    annulerCommande.annuler();
}
```

Lorsque le bouton Annuler est enfoncé, nous invoquons la méthode undo() de la commande stockée dans undoCommand. Cela annule l'opération de la dernière commande exécutée.

```
chaîne publique toString() {
    // code toString ici...
}
```

Mise à jour pour ajouter des commandes d'annulation.

```
}
```

Il est temps de vérifier la qualité de ce bouton Annuler !

Ok, retravaillons un peu le harnais de test pour tester le bouton d'annulation :

classe publique RemoteLoader {

```
public static void main(String[] args) {
    RemoteControlWithUndo remoteControl = nouveau RemoteControlWithUndo();

    Lumière livingRoomLight = new Light("Salon");
    Commande LightOn livingRoomOn =
        nouvelle commande LightOn(livingRoomLight);
    commande LightOff livingRoomOff =
        nouvelle commande LightOffCommand(livingRoomLight);

    remoteControl.setCommand(0, éclairage de la salle de séjour allumé, éclairage de la salle de séjour éteint);

    remoteControl.onButtonWasPushed(0);
    remoteControl.offButtonWasPushed(0);
    System.out.println(remoteControl);
    remoteControl.undoButtonWasPushed();
    remoteControl.offButtonWasPushed(0);
    remoteControl.onButtonWasPushed(0);
    System.out.println(remoteControl);
    remoteControl.undoButtonWasPushed();
}
```

Créez une lumière et nos nouvelles commandes d'activation et de désactivation de la lumière activées par undo().

Ajoutez les commandes d'éclairage à la télécommande dans l'emplacement 0.

Allumez la lumière, puis éteignez-la, puis annulez-la.

Ensuite, éteignez la lumière, rallumez-la et annulez-la.

Et voici les résultats des tests...

```
Fenêtre d'édition de fichier Aide AnnulerCommandesDéfierEntropy
% Chargeur à distance Java
La lumière est allumée ← Allumez la lumière, puis éteignez-la.
La lumière est éteinte ←

----- Télécommande -----
[emplACEMENT 0] LightOnCommand ← Voici les commandes Lumière.
[emplACEMENT 1] NoCommand
[emplACEMENT 2] NoCommand
[emplACEMENT 3] NoCommand
[emplACEMENT 4] NoCommand
[emplACEMENT 5] NoCommand
[emplACEMENT 6] NoCommand
[annuler] Commande LightOff ← Annuler maintenant la commande
                                LightOffCommand, la dernière commande invoquée.

La lumière est allumée ← La touche Annuler a été enfoncée... la commande
                                LightOffCommand undo() rallume la lumière.
La lumière est éteinte ← Ensuite, nous éteignons la lumière et la rallumons.

----- Télécommande -----
[emplACEMENT 0] LightOnCommand ←
[emplACEMENT 1] NoCommand
[emplACEMENT 2] NoCommand
[emplACEMENT 3] NoCommand
[emplACEMENT 4] NoCommand
[emplACEMENT 5] NoCommand
[emplACEMENT 6] NoCommand
[annuler] LightOnCommand ← Annuler contient désormais LightOnCommand, la dernière
                                commande invoquée.

La lumière est éteinte ← La touche Annuler a été enfoncée, donc la lumière est à nouveau éteinte.
```

Utilisation de l'état pour implémenter l'annulation

Ok, l'implémentation de l'annulation sur la lumière était instructive mais un peu trop facile. En général, nous devons gérer un peu d'état pour implémenter l'annulation. Essayons quelque chose d'un peu plus intéressant, comme le CeilingFan des classes vendor. La classe CeilingFan permet de définir un certain nombre de vitesses avec une méthode off.

Voici le code source de la classe CeilingFan :

```

classe publique CeilingFan {
    public static final int HIGH = 3; public static
    final int MEDIUM = 2; public static final int
    LOW = 1; public static final int OFF = 0;
    Emplacement de la chaîne;

    vitesse int;

    ventilateur de plafond public (emplacement de la chaîne) {
        this.location = emplacement ;
        vitesse = OFF ;
    }

    public void haut() {
        vitesse = ÉLEVÉE;
        // code pour régler le ventilateur sur haut
    }

    public void medium() {
        vitesse = MOYENNE;
        // code pour régler le ventilateur sur moyen
    }

    public void faible() {
        vitesse = FAIBLE;
        // code pour régler le ventilateur sur faible
    }

    public void off() {
        vitesse = OFF;
        // code pour éteindre le ventilateur
    }

    public int getSpeed() {
        vitesse de retour;
    }

```



Notez que la classe CeilingFan contient l'état local représentant la vitesse du ventilateur de plafond.

Hmm, donc pour implémenter correctement l'annulation, je devrais prendre en compte la vitesse précédente du ventilateur de plafond...

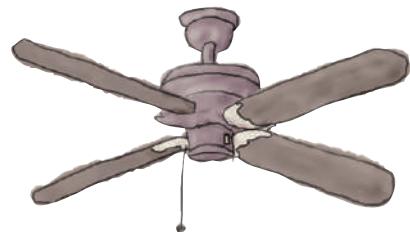


Ces méthodes permettent de régler la vitesse du ventilateur de plafond.

Nous pouvons obtenir la vitesse actuelle du ventilateur de plafond en utilisant getSpeed().

Ajout d'Annuler aux commandes du ventilateur de plafond

Passons maintenant à l'ajout d'une annulation aux différentes commandes Ceiling Fan. Pour ce faire, nous devons suivre le dernier réglage de vitesse du ventilateur et, si la méthode undo() est appelée, restaurer le ventilateur à son réglage précédent. Voici le code de la commande CeilingFanHighCommand :



la classe publique CeilingFanHighCommand implémente Command {

Ventilateur de plafond Ventilateur de plafond;

```
int prevSpeed;
```

```
public CeilingFanHighCommand(CeilingFan ceilingFan) {  
    ceci.ceilingFan = ventilateur de plafond;  
}
```

```
public void exécuter() {  
    prevSpeed = ceilingFan.getSpeed();  
    ventilateurdeplafond.haut();  
}
```

```
public void annuler() {  
    si (prevSpeed == CeilingFan.HIGH) {  
        ventilateurdeplafond.haut();  
    } sinon si (prevSpeed == CeilingFan.MEDIUM) {  
        ventilateur de plafond.medium();  
    } sinon si (prevSpeed == CeilingFan.LOW) {  
        ventilateur de plafond.low();  
    } sinon si (prevSpeed == CeilingFan.OFF) {  
        ventilateur de plafond.off();  
    }  
}
```

Nous avons ajouté un état local pour suivre la vitesse précédente du ventilateur.

Dans execute(), avant de modifier la vitesse du ventilateur, nous devons d'abord enregistrer son état précédent, juste au cas où nous aurions besoin d'annuler nos actions.

Pour annuler, nous remettons la vitesse du ventilateur à sa vitesse précédente.



Il nous reste trois commandes de ventilateur de plafond à écrire : faible, moyen et éteint. Voyez-vous comment elles sont implémentées ?

Préparez-vous à tester le ventilateur de plafond

Il est temps de charger notre télécommande avec les commandes du ventilateur de plafond. Nous allons charger le bouton marche de l'emplacement 0 avec le réglage moyen pour le ventilateur et l'emplacement 1 avec le réglage élevé. Les deux boutons d'arrêt correspondants maintiendront la commande d'arrêt du ventilateur de plafond.

Voici notre script de test :

```
classe publique RemoteLoader {
```

```
    public static void main(String[] args) {
        RemoteControlWithUndo remoteControl = nouveau RemoteControlWithUndo();
```

```
        CeilingFan ceilingFan = new CeilingFan("Salon");
```

```
        Commande CeilingFanMedium ceilingFanMedium =
```

```
            nouvelle commande CeilingFanMediumCommand(ceilingFan);
```

```
        CeilingFanHighCommand ceilingFanHigh =
```

```
            nouvelle commande CeilingFanHighCommand(ceilingFan);
```

```
        commande CeilingFanOff ceilingFanOff =
```

```
            nouvelle commande CeilingFanOffCommand(ceilingFan);
```

```
        remoteControl.setCommand(0, ceilingFanMedium, ceilingFanOff);
```

```
        remoteControl.setCommand(1, ceilingFanHigh, ceilingFanOff);
```

```
        remoteControl.onButtonWasPushed(0);
```

```
        remoteControl.offButtonWasPushed(0);
```

```
        System.out.println(remoteControl);
```

```
        remoteControl.undoButtonWasPushed();
```

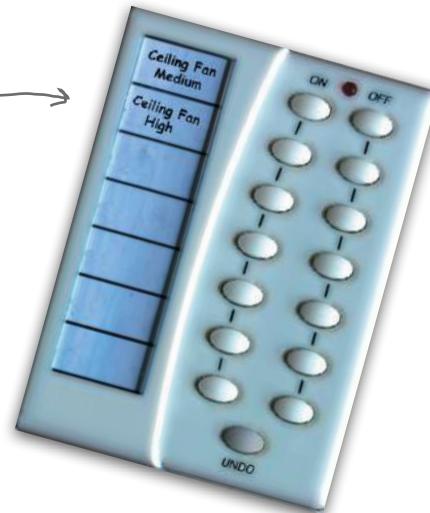
```
        remoteControl.onButtonWasPushed(1);
```

```
        System.out.println(remoteControl);
```

```
        remoteControl.undoButtonWasPushed();
```

```
}
```

```
}
```



Ici, nous instancions trois commandes : moyen, élevé et désactivé.

Ici, nous mettons le medium dans l'emplacement 0 et le high dans l'emplacement 1. Nous chargeons également la commande off.

Tout d'abord, allumez le ventilateur à vitesse moyenne. Puis éteignez-le.

Annuler ! Il devrait revenir à un niveau moyen...

Cette fois, mettez-le à fond.

Et, encore une annulation ; il devrait revenir à moyen.

Test du ventilateur de plafond...

Ok, allumons la télécommande, chargeons-la de commandes et appuyons sur quelques boutons !

VFenêtre d'édition de fichier Aide Annuler !

% Chargeur à distance Java

Le ventilateur de plafond du salon est à feu moyen. Le ventilateur de plafond du salon est éteint.

← Allumez le ventilateur de plafond à puissance moyenne, puis éteignez-le.

----- Télécommande ----- [emplacement 0] CeilingFanMediumCommand [emplacement 1] CeilingFanHighCommand [emplacement 2] NoCommand [emplacement 3] NoCommand [emplacement 4] NoCommand [emplacement 5] NoCommand [emplacement 6] NoCommand [annuler] Commande de désactivation du ventilateur de plafond

Commande d'arrêt du ventilateur de plafond Commande d'arrêt du ventilateur de plafond Pas de commande ← Voici les commandes de la télécommande...

... et annuler a exécuté la dernière commande, CeilingFanOffCommand, avec la vitesse moyenne précédente.

Le ventilateur de plafond du salon est réglé sur moyen. Le ventilateur de plafond du salon est réglé sur élevé.

← Annulez la dernière commande et le niveau revient au niveau moyen.
← Maintenant, allumez-le à fond.

----- Télécommande ----- [emplacement 0] CeilingFanMediumCommand [emplacement 1] CeilingFanHighCommand [emplacement 2] NoCommand [emplacement 3] NoCommand [emplacement 4] NoCommand [emplacement 5] NoCommand [emplacement 6] NoCommand [annuler] CeilingFanHighCommand

Commande d'arrêt du ventilateur de plafond Commande d'arrêt du ventilateur de plafond Pas de commande Pas de commande Pas de commande Pas de commande ← Maintenant, high est la dernière commande exécutée.

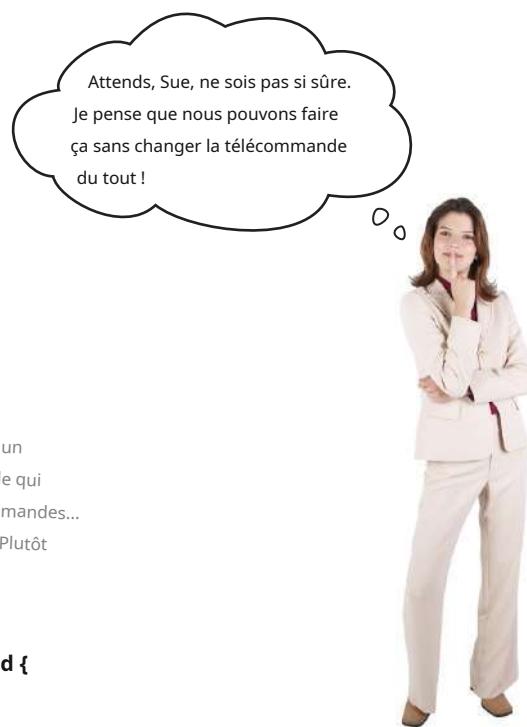
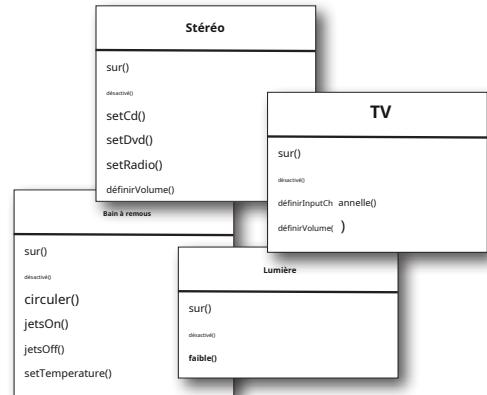
Le ventilateur de plafond du salon est réglé à moyen

← Encore une annulation et le ventilateur de plafond revient à vitesse moyenne.

%

Chaque télécommande a besoin d'un mode Fête !

Quel est l'intérêt d'avoir une télécommande si vous ne pouvez pas appuyer sur un bouton pour tamiser les lumières, allumer la chaîne stéréo et la télévision, et faire fonctionner le jacuzzi ?



L'idée de Mary est de créer un nouveau type de commande qui peut exécuter d'autres commandes... et plus d'une d'entre elles ! Plutôt bonne idée, non ?

classe publique MacroCommand implémente Command {
Commandes Command[] ;

```

public MacroCommand(Command[] commandes) {
    this.commands = commandes;
}
  
```

Prenez un tableau de commandes et stockez-les dans la macrocommande.

```

public void exécuter() {
    pour (int i = 0; i < commandes.length; i++) {
        commandes[i].execute();
    }
}
  
```

Lorsque la macro est exécutée par la télécommande,
exécutez ces commandes une par une.

Utilisation d'une macro-commande

Voyons comment utiliser une macro-commande :

- 1 Nous créons d'abord l'ensemble des commandes que nous souhaitons intégrer à la macro :

```
Lumière lumière = nouvelle Lumière("Salon"); TV  
tv = nouvelle TV("Salon");  
Stéréo stéréo = new Stereo("Salon"); Hottub hottub =  
new Hottub();
```

Créez tous les appareils : une lumière, une télévision, une chaîne stéréo et un jacuzzi.

```
LightOnCommand lightOn = new LightOnCommand(lumière);  
StereoOnCommand stereoOn = new StereoOnCommand(stéréo);  
TVOnCommand tvOn = new TVOnCommand(tv);  
HottubOnCommand hottubOn = nouveau HottubOnCommand(hottub);
```

Créez maintenant toutes les commandes On pour les contrôler.



Sharpen your pencil

Nous aurons également besoin de commandes pour les boutons d'arrêt. Écrivez le code pour les créer ici :

- 2 Ensuite, nous créons deux tableaux, un pour les commandes On et un pour les commandes Off, et les chargeons avec les commandes correspondantes :

```
Commande[] partyOn = { lightOn, stereoOn, tvOn, hottubOn}; Commande[]  
partyOff = { lightOff, stereoOff, tvOff, hottubOff};
```

Créez un tableau pour les commandes On et un tableau pour les commandes Off...

```
MacroCommand partyOnMacro = new MacroCommand(partyOn);  
MacroCommand partyOffMacro = new MacroCommand(partyOff);
```

... et créer deux macros correspondantes pour les tenir.

- 3 Ensuite, nous attribuons une MacroCommand à un bouton comme nous le faisons toujours :

```
remoteControl.setCommand(0, partyOnMacro, partyOffMacro);
```

Affecter la macro

commandez un bouton comme vous le feriez pour n'importe quelle commande.

4

Finalement, il ne nous reste plus qu'à appuyer sur quelques boutons et voir si cela fonctionne.

```
System.out.println(remoteControl);
System.out.println("--- Enclenchement de la macro---");
remoteControl.onButtonWasPushed(0);
System.out.println("--- Désactivation de la macro---");
remoteControl.offButtonWasPushed(0);
```

Voici le résultat.

Fenêtre d'édition de fichier Aide Vous ne pouvez pas battre ABabka

```
% Chargeur à distance Java
----- - Télécommande -----
[emplacement 0] MacroCommande macro
[emplacement 1] NoCommand      Pas de commande
[emplacement 2] NoCommand      Pas de commande
[emplacement 3] NoCommand      Pas de commande
[emplacement 4] NoCommand      Pas de commande
[emplacement 5] NoCommand      Pas de commande
[emplacement 6] NoCommand      Pas de commande
[annuler] NoCommand

--- En appuyant sur la macro, la
lumière est allumée
La chaîne stéréo du salon est allumée. La
télévision du salon est allumée.
La chaîne de télévision du salon est réglée sur DVD. Le
jacuzzi chauffe à une température de 104 degrés. Le jacuzzi
bouillonne !
--- Pousser la macro hors tension --- La
lumière est éteinte
La chaîne stéréo du salon est éteinte. La
télévision du salon est éteinte. Le jacuzzi
refroidit à 98 degrés.
```

Voici les deux commandes macro.

Toutes les commandes de la macro sont exécutées lorsque nous invoquons la macro on...

... et lorsque nous invoquons la macro off. On dirait que ça marche.



La seule chose qui manque à notre MacroCommand est sa fonctionnalité d'annulation. Lorsque le bouton Annuler est enfoncé après une macro-commande, toutes les commandes qui ont été invoquées dans la macro doivent annuler leurs actions précédentes. Voici le code de MacroCommand ; allez-y et implémentez la méthode undo() :

```
classe publique MacroCommand implémente Command {  
    Commandes Command[];  
  
    public MacroCommand(Command[] commandes) {  
        this.commands = commandes;  
    }  
  
    public void exécuter() {  
        pour (int i = 0; i < commandes.length; i++) {  
            commandes[i].execute();  
        }  
    }  
  
    public void annuler() {  
        // Implémentez ici la logique d'annulation  
    }  
}
```

there are no
Dumb Questions

Q:

Ai-je toujours besoin d'un récepteur ?

Pourquoi l'objet de commande ne peut-il pas implémenter les détails de la méthode execute() ?

UN:

En général, nous nous efforçons d'utiliser des objets de commande « stupides » qui invoquent simplement une action sur un récepteur. Cependant, il existe de nombreux exemples d'objets de commande « intelligents » qui implémentent la plupart, voire la totalité, de la logique nécessaire pour exécuter une requête. Vous pouvez certainement le faire ; gardez simplement à l'esprit que vous n'aurez plus le même niveau de découplage entre l'invocateur et le récepteur, et que vous ne pourrez pas non plus paramétriser vos commandes avec des récepteurs.

Q:

Comment puis-je mettre en place un historique des opérations d'annulation ? En d'autres termes, je souhaite pouvoir appuyer plusieurs fois sur le bouton Annuler.

UN:

Excellent question. C'est assez simple en fait ; au lieu de conserver simplement une référence à la dernière commande exécutée, vous conservez une pile de commandes précédentes. Ensuite, chaque fois que vous appuyez sur Annuler, votre appelant retire le premier élément de la pile et appelle sa méthode undo().

Q:

Aurais-je pu simplement implémenter le mode fête en tant que commande en créant une PartyCommand et en plaçant les appels pour exécuter les autres commandes dans la méthode execute() de PartyCommand ?

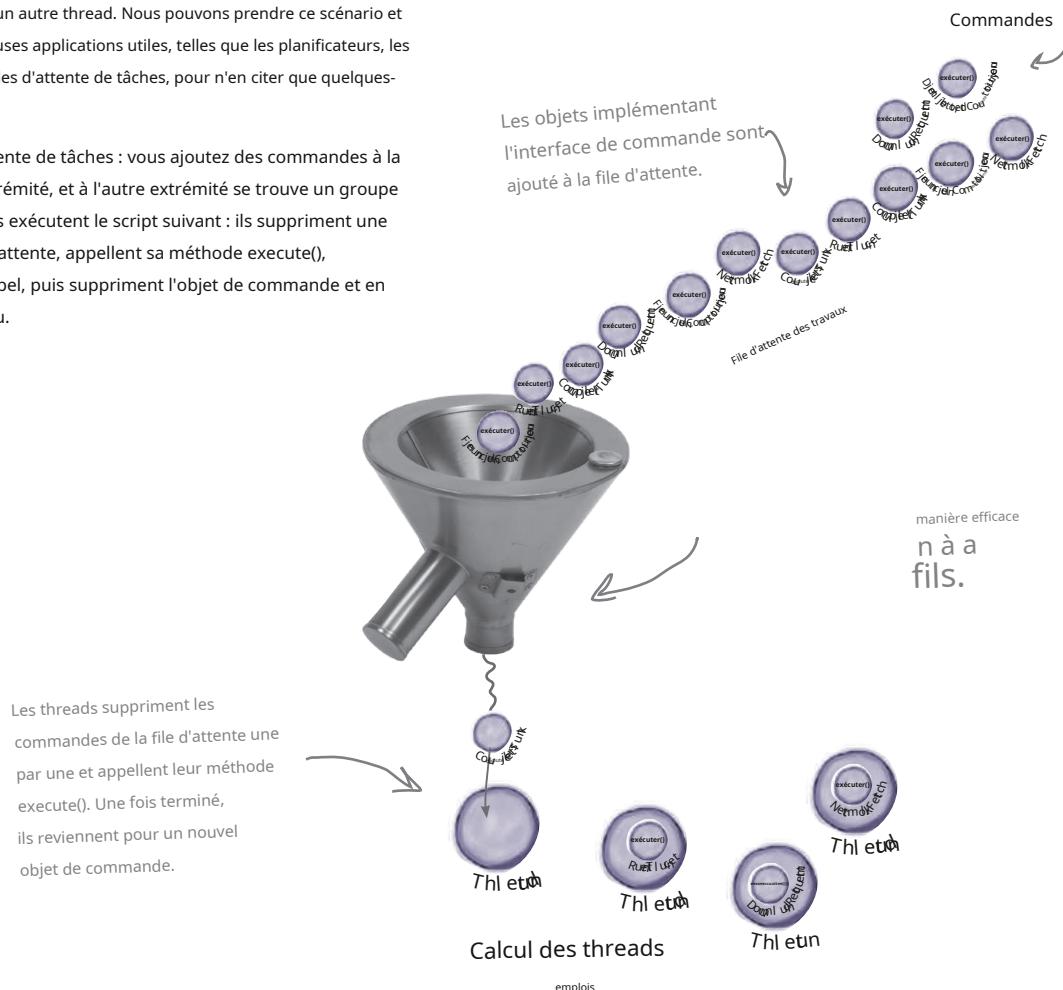
UN:

Vous pourriez le faire, mais vous auriez essentiellement à « coder en dur » le mode fête dans PartyCommand. Pourquoi se donner la peine ? Avec MacroCommand, vous pouvez décider de manière dynamique quelles commandes vous souhaitez intégrer dans PartyCommand, ce qui vous donne plus de flexibilité dans l'utilisation des MacroCommands. En général, MacroCommand est une solution plus élégante et nécessite moins de nouveau code.

Autres utilisations du modèle de commande : mise en file d'attente des requêtes

Les commandes nous permettent de regrouper un élément de calcul (un récepteur et un ensemble d'actions) et de le transmettre sous forme d'objet de première classe. Désormais, le calcul lui-même peut être invoqué longtemps après qu'une application cliente a créé l'objet de commande. En fait, il peut même être invoqué par un autre thread. Nous pouvons prendre ce scénario et l'appliquer à de nombreuses applications utiles, telles que les planificateurs, les pools de threads et les files d'attente de tâches, pour n'en citer que quelques-unes.

Imaginez une file d'attente de tâches : vous ajoutez des commandes à la file d'attente à une extrémité, et à l'autre extrémité se trouve un groupe de threads. Les threads exécutent le script suivant : ils suppriment une commande de la file d'attente, appellent sa méthode execute(), attendent la fin de l'appel, puis suppriment l'objet de commande et en récupèrent un nouveau.



Notez que les classes de la file d'attente de tâches sont totalement découplées des objets qui effectuent le calcul. Une minute, un thread peut effectuer un calcul financier, et la minute suivante, il peut récupérer quelque chose du réseau. Les objets de la file d'attente de tâches ne s'en soucient pas ; ils récupèrent simplement les commandes et appellent execute(). De même, tant que vous placez des objets dans la file d'attente qui implémentent le modèle de commande, votre méthode execute() sera invoquée lorsqu'un thread sera disponible.



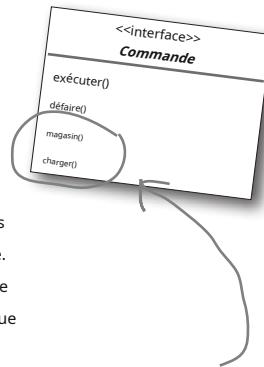
Comment un serveur Web pourrait-il exploiter une telle file d'attente ? À quelles autres applications pouvez-vous penser ?

Autres utilisations du modèle de commande : journalisation des requêtes

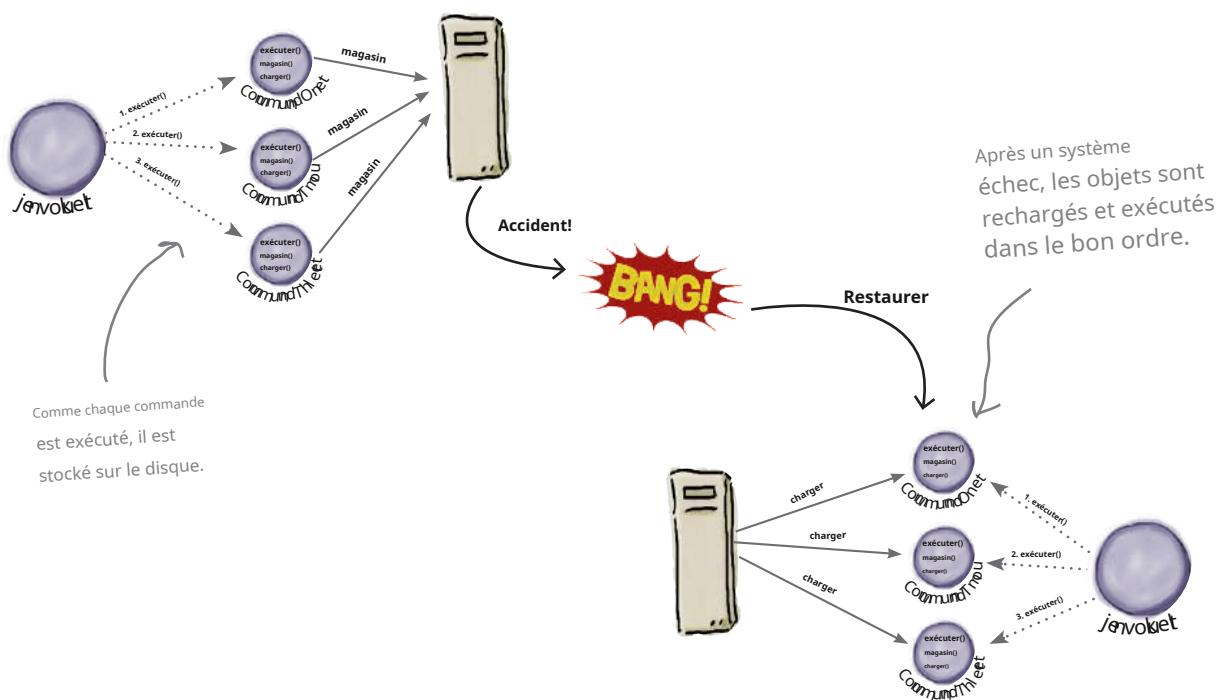
La sémantique de certaines applications nécessite que nous enregistriions toutes les actions et que nous soyons en mesure de récupérer après un crash en réinvoquant ces actions. Le modèle de commande peut prendre en charge cette sémantique avec l'ajout de deux méthodes : `store()` et `load()`. En Java, nous pourrions utiliser la sérialisation d'objets pour implémenter ces méthodes, mais les réserves normales concernant l'utilisation de la sérialisation pour la persistance s'appliquent.

Comment cela fonctionne-t-il ? Lorsque nous exécutons des commandes, nous stockons un historique de celles-ci sur le disque. Lorsqu'un crash se produit, nous rechargeons les objets de commande et appelons leurs méthodes `execute()` par lots et dans l'ordre.

Ce type de journalisation n'aurait aucun sens pour une commande à distance. Cependant, de nombreuses applications invoquent des actions sur des structures de données volumineuses qui ne peuvent pas être enregistrées rapidement à chaque fois qu'une modification est apportée. En utilisant la journalisation, nous pouvons enregistrer toutes les opérations depuis le dernier point de contrôle et, en cas de défaillance du système, appliquer ces opérations à notre point de contrôle. Prenons par exemple une application de tableau : nous pourrions vouloir implémenter notre récupération après défaillance en enregistrant les actions sur la feuille de calcul plutôt qu'en écrivant une copie de la feuille de calcul sur le disque à chaque fois qu'une modification se produit. Dans des applications plus avancées, ces techniques peuvent être étendues pour s'appliquer à des ensembles d'opérations de manière transactionnelle afin que toutes les opérations soient terminées ou qu'aucune d'entre elles ne le soit.



Nous ajoutons deux méthodes de journalisation.



Modèle de commande dans le monde réel

Vous vous souvenez de la petite application qui a changé votre vie au chapitre 2 ?

Dans ce chapitre, nous avons vu comment la bibliothèque Swing de Java regorge d'observateurs sous la forme d'ActionListeners qui écoutent (ou *observer*) événements sur les composants de l'interface utilisateur.

Eh bien, il s'avère qu' ActionListener n'est pas seulement une interface d'observateur, c'est aussi une interface de commande, et nos classes AngelListener et DevilListener ne sont pas seulement des observateurs, mais aussi des commandes concrètes. C'est vrai, nous avons deux modèles dans un seul exemple !



Voici le code (les parties importantes en tout cas) de la petite application qui change la vie du chapitre 2. Voyez si vous pouvez identifier qui est le client, qui sont les commandes, qui est l'appelant et qui est le récepteur.

```
classe publique SwingObserverExample {
    // Installation ...
    JButton button = new JButton("Dois-je le faire ?");
    button.addActionListener(new AngelListener());
    button.addActionListener(new DevilListener()); // Définir les
    propriétés du cadre ici
}
la classe AngelListener implémente ActionListener {
    public void actionPerformed(événement ActionEvent) {
        System.out.println("Ne le faites pas, vous pourriez le regretter !");
    }
}
la classe DevilListener implémente ActionListener {
    public void actionPerformed(événement ActionEvent) {
        System.out.println("Allez, fais-le !");
    }
}
```



Sharpen your pencil Solution

Voici le code (les éléments importants en tout cas) de la petite application qui change la vie du chapitre 2. Voyez si vous pouvez identifier qui est le client, qui sont les commandes, qui est l'appelant et qui est le récepteur ?

Voici notre solution.

```
classe publique SwingObserverExample {
```

```
    // Installation ...
```

```
        JButton button = new JButton("Dois-je le faire ?");
        button.addActionListener(new AngelListener());
        button.addActionListener(new DevilListener());
```

Le bouton est notre Invoker. Le bouton appelle les méthodes actionPerformed() (comme execute()) dans les commandes (les ActionListeners) lorsque vous cliquez sur le bouton.

```
    // Définir les propriétés du cadre ici
```

```
}
```

Le Client est la classe qui configure les composants Swing et définit les commandes (AngelListener et DevilListener) dans l'Invoker (le bouton).

```
la classe AngelListener implémente ActionListener {
```

```
    public void actionPerformed(événement ActionEvent) {
        System.out.println("Ne le faites pas, vous pourriez le regretter !");
    }
}
```

ActionListener est l'interface de commande : elle possède une méthode, actionPerformed(), qui, comme execute(), est exécutée lorsque la commande est invoquée.

```
la classe DevilListener implémente ActionListener {
```

```
    public void actionPerformed(événement ActionEvent) {
        System.out.println("Allez, fais-le !");
    }
}
```

AngelListener et DevilListener sont nos commandes concrètes. Elles implémentent l'interface de commande (dans ce cas, ActionListener).

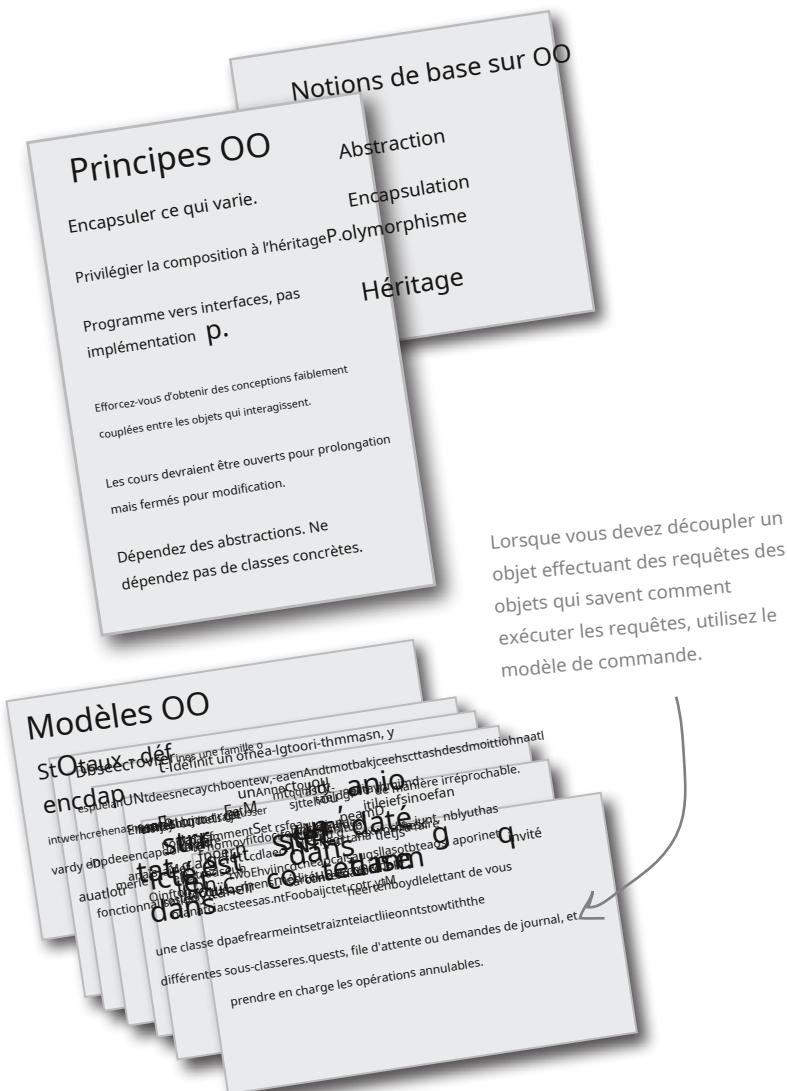
```
}
```

Dans cet exemple, le récepteur est l'objet système. N'oubliez pas que l'appel d'une commande entraîne des actions sur le récepteur. Dans une application Swing classique, cela entraînerait l'appel d'actions sur d'autres composants de l'interface utilisateur.



Des outils pour votre boîte à outils de conception

Votre boîte à outils commence à s'alourdir ! Dans ce chapitre, nous avons ajouté un modèle qui nous permet d'encapsuler des méthodes dans des objets Command : les stocker, les transmettre et les invoquer lorsque vous en avez besoin.



BULLET POINTS

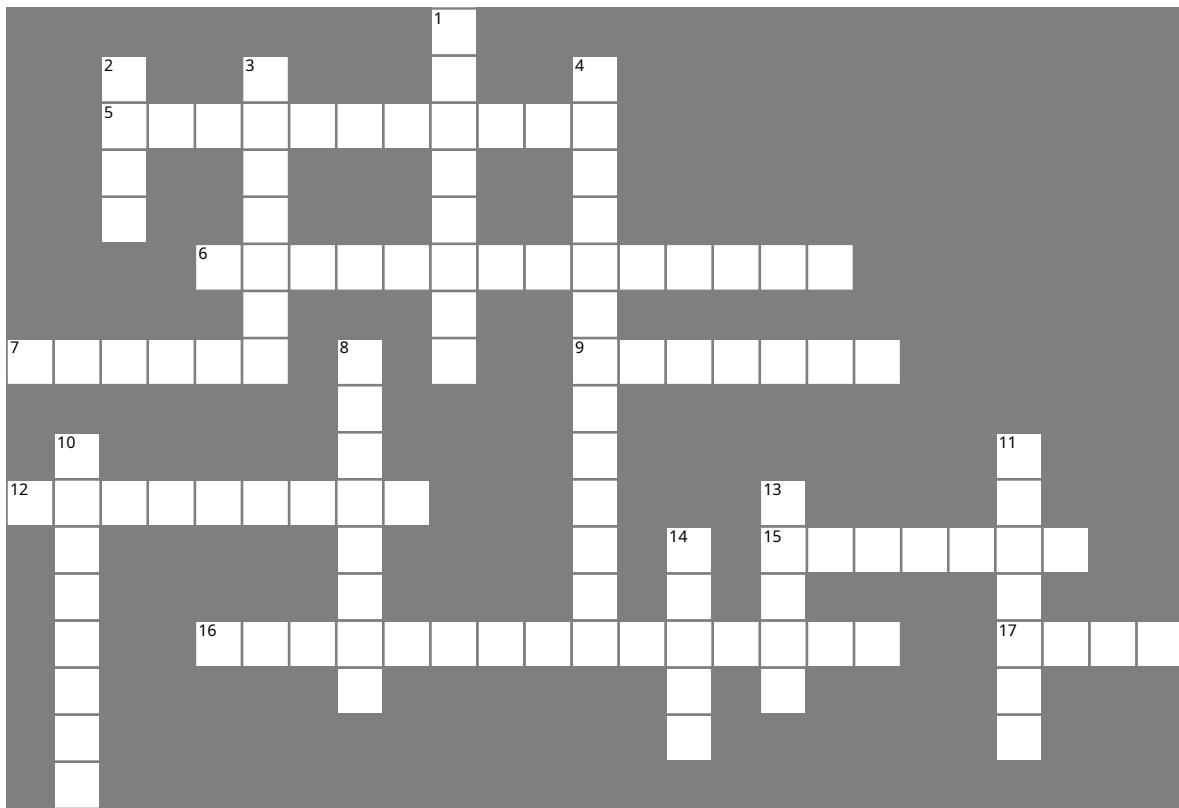
- f Le modèle de commande découpe un objet faisant une requête de celui qui sait comment l'exécuter.
- f Un objet Command est au centre de ce découplage et encapsule un récepteur avec une action (ou un ensemble d'actions).
- f Un appelant fait une demande à un objet Command en appelant sa méthode execute(), qui appelle ces actions sur le récepteur.
- f Les invokeurs peuvent être paramétrés avec des commandes, même dynamiquement lors de l'exécution.
- f Les commandes peuvent prendre en charge annuler en implementant une méthode undo() qui restaure l'objet à son état précédent avant le dernier appel de la méthode execute().
- f Les macrocommandes sont une extension simple du modèle de commande qui permettent d'invoquer plusieurs commandes. De même, les macrocommandes peuvent facilement prendre en charge undo().
- f Dans la pratique, il n'est pas rare que des personnes « intelligentes » Objets de commande à mettre en œuvre la demande eux-mêmes plutôt que déléguer à un récepteur.
- f Les commandes peuvent également être utilisées pour implémenter des systèmes de journalisation et de transaction.



Mots croisés sur les modèles de conception

Il est temps de prendre une pause et de laisser tout cela pénétrer.

C'est un autre jeu de mots croisés ; tous les mots solutions proviennent de ce chapitre.

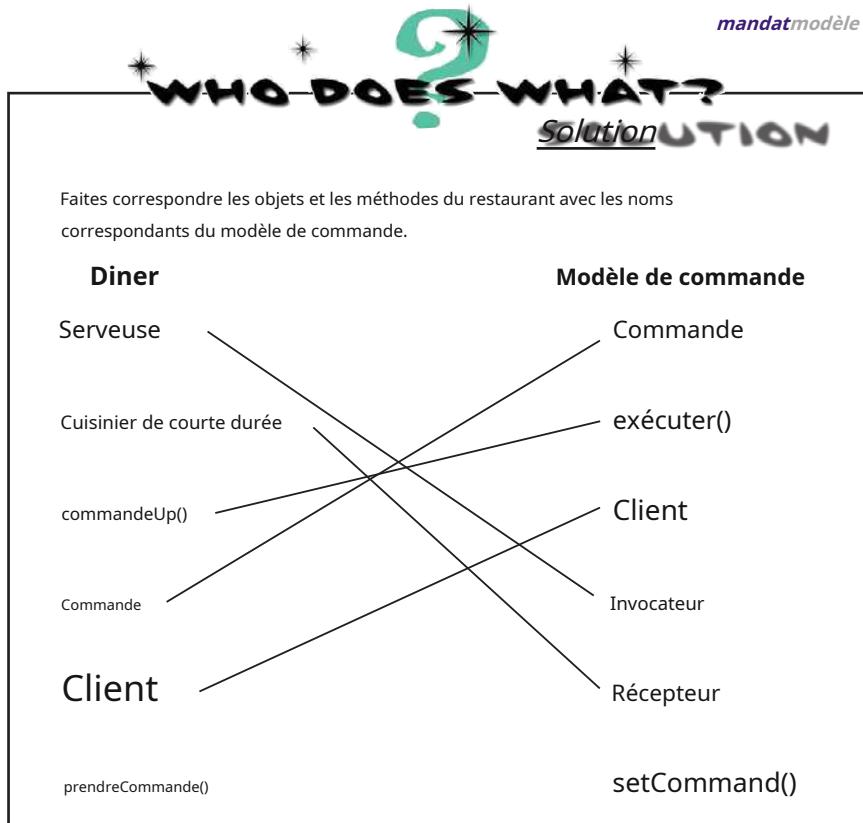


À TRAVERS

5. Notre ville préférée.
 6. Entreprise qui nous a permis de faire du bouche-à-oreille.
 7. Rôle du client dans le modèle de commande.
 9. Objet qui connaît les actions et le récepteur.
 12. L'invocateur et le récepteur sont _____.
15. La serveuse en était une.
 16. La nourriture du restaurant Dr. Seuss (quatre mots).
 17. Une autre chose que Command peut faire.

VERS | F BAS

1. Le cuisinier et cette personne étaient définitivement découplés.
 2. La serveuse n'a pas fait ça.
 3. Une commande encapsule ceci.
 4. Agissez comme les récepteurs de la télécommande (deux mots).
 8. Objet qui sait faire avancer les choses.
 10. Exécute une demande.
 11. Toutes les commandes fournissent cela.
 13. Notre premier objet de commande contrôlait ceci.
 14. Une commande _____ un ensemble d'actions et un récepteur.



Sharpen your pencil Solution

Voici le code de la classe GarageDoorOpenCommand.

```
la classe publique GarageDoorOpenCommand implémente Command {
    Porte de garage Porte de garage;

    public GarageDoorOpenCommand(GarageDoor garageDoor) {
        ceci.portedegarage = portedegarage;
    }
    public void exécuter() {
        portedegarage.up();
    }
}
```

Voici le résultat :

Aide sur la fenêtre d'édition de fichiers GreenEggs&Ham

```
% java RemoteControlTest
```

La lumière est allumée

La porte du garage est ouverte

%



Exercise Solution

Voici la méthode undo() pour le MacroCommande.

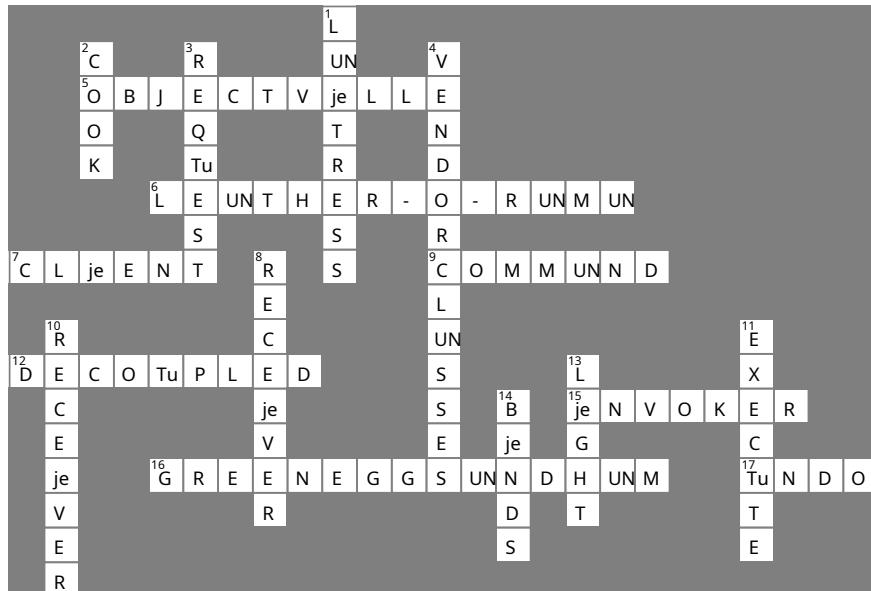
```
classe publique MacroCommand implémente Command {  
    Commandes Command[] ;  
    public MacroCommand(Command[] commandes) {  
        this.commands = commandes;  
    }  
    public void exécuter() {  
        pour (int i = 0; i < commandes.length; i++) {  
            commandes[i].execute();  
        }  
    }  
    public void annuler() {  
        pour (int i = commandes.length - 1; i >= 0; i--) {  
            commandes[i].undo();  
        }  
    }  
}
```



Sharpen your pencil Solution

Voici le code pour créer des commandes pour le bouton d'arrêt.

```
LightOffCommand lightOff = new LightOffCommand(light);
StereoOffCommand stereoOff = new StereoOffCommand(stéreo);
TVOffCommand tvOff = new TVOffCommand(tv);
HottubOffCommand hottubOff = nouveau HottubOffCommand(hottub);
```



Les modèles d'adaptateur et de façade



Être adaptable

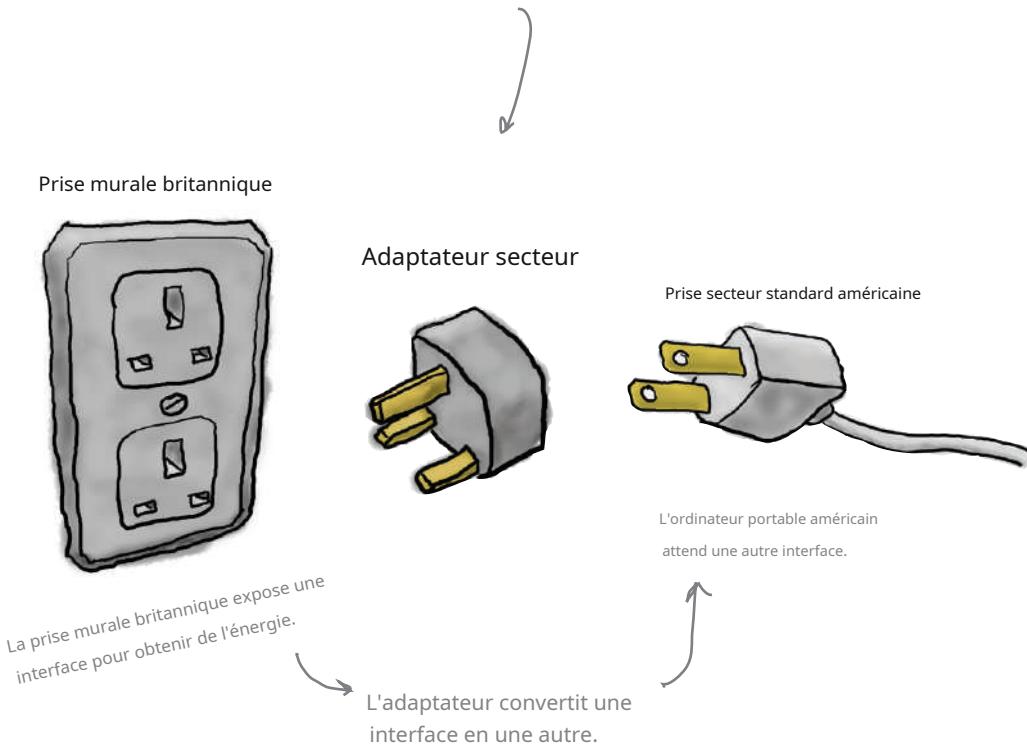


Dans ce chapitre, nous allons tenter des exploits impossibles comme insérer une cheville carrée dans un trou rond. Cela semble impossible ? Pas quand

nous avons des modèles de conception. Vous vous souvenez du modèle Decorator ? Nousobjets emballéspour leur donner de nouvelles responsabilités. Nous allons maintenant envelopper certains objets dans un but différent : faire en sorte que leurs interfaces ressemblent à quelque chose qu'elles ne sont pas. Pourquoi ferions-nous cela ? Pour pouvoir adapter une conception qui attend une interface à une classe qui implémente une interface différente. Ce n'est pas tout ; tant que nous y sommes, nous allons examiner un autre modèle qui enveloppe des objets pour simplifier leur interface.

Des adaptateurs tout autour de nous

Vous n'aurez aucun mal à comprendre ce qu'est un adaptateur OO, car le monde réel en regorge. Par exemple : avez-vous déjà eu besoin d'utiliser un ordinateur portable fabriqué aux États-Unis en Grande-Bretagne ? Dans ce cas, vous avez probablement eu besoin d'un adaptateur secteur...



Vous savez à quoi sert l'adaptateur : il se place entre la prise de votre ordinateur portable et la prise secteur britannique ; son rôle est d'adapter la prise britannique pour que vous puissiez y brancher votre ordinateur portable et recevoir du courant. Ou voyez les choses de cette façon : l'adaptateur change l'interface de la prise en une interface que votre ordinateur portable attend.

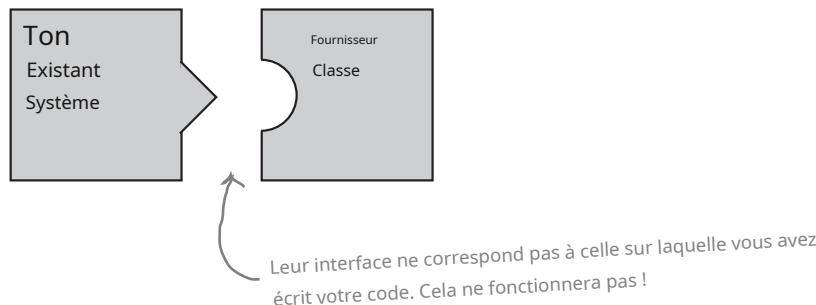
Certains adaptateurs secteur sont simples : ils modifient simplement la forme de la prise pour qu'elle corresponde à votre fiche et ils laissent passer le courant secteur directement à travers, mais d'autres adaptateurs sont plus complexes en interne et peuvent nécessiter d'augmenter ou de diminuer la puissance pour répondre aux besoins de vos appareils.

Bon, c'est le monde réel. Qu'en est-il des adaptateurs orientés objet ? Eh bien, nos adaptateurs OO jouent le même rôle que leurs homologues du monde réel : ils prennent une interface et l'adaptent à celle qu'attend un client.

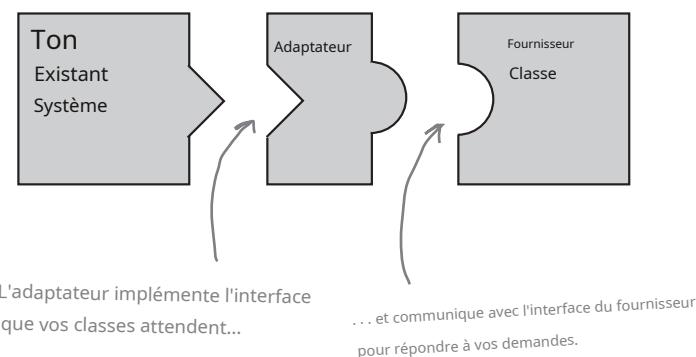
À combien d'autres adaptateurs du monde réel pouvez-vous penser ?

Adaptateurs orientés objet

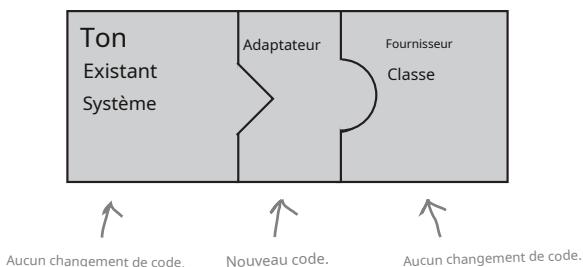
Supposons que vous disposez d'un système logiciel existant dans lequel vous devez intégrer une nouvelle bibliothèque de classes de fournisseur, mais que le nouveau fournisseur a conçu ses interfaces différemment du fournisseur précédent :



D'accord, vous ne voulez pas résoudre le problème en modifiant votre code existant (et vous ne pouvez pas modifier le code du fournisseur). Alors que faire ? Eh bien, vous pouvez écrire une classe qui adapte la nouvelle interface du fournisseur à celle que vous attendez.



L'adaptateur agit comme intermédiaire en recevant les demandes du client et en les convertissant en demandes qui ont du sens sur les classes du fournisseur.



Pouvez-vous penser à une solution qui ne vous oblige pas à écrire de code supplémentaire pour intégrer les nouvelles classes du fournisseur ? Pourquoi ne pas demander au fournisseur de fournir la classe d'adaptateur ?

Si ça marche comme un canard et cancane comme un canard, alors ça doit être une ~~dinde~~-canard enveloppée dans un adaptateur de canard...

Il est temps de voir un adaptateur en action. Vous vous souvenez de nos canards du chapitre 1 ? Passons en revue une version légèrement simplifiée des interfaces et des classes Duck :

```
interface publique Canard {  
    public void charlatan();  
    public void voler();  
}
```

Cette fois-ci, nos canards implémentent une interface Duck qui permet Des canards qui cancanent et volent.



Voici une sous-classe de canard, le canard colvert :

```
la classe publique MallardDuck implémente Duck {  
    public void charlatan() {  
        System.out.println("Coin-coin");  
    }  
  
    public void voler() {  
        System.out.println("Je vole");  
    }  
}
```

Implémentations simples : MallardDuck imprime simplement ce qu'il fait.

Il est maintenant temps de rencontrer la nouvelle volaille du quartier :

```
interface publique Turquie {  
    public void engloutir();  
    public void voler();  
}
```

Les dindes ne cancanent pas, elles glougloutissent.
Les dindes peuvent voler, mais seulement sur de courtes distances.

la classe publique WildTurkey implémente la Turquie {

public void engloutir() {

System.out.println("Glouglouter glouglouter");

}

Voici une implémentation concrète de la Turquie ; comme MallardDuck, il imprime simplement ses actions.

public void voler() {

System.out.println("Je vole sur une courte distance");

}

}

Maintenant, disons que vous manquez d'objets Canard et que vous souhaitez utiliser des objets Dinde à leur place.

Évidemment, nous ne pouvons pas utiliser les dindes directement car elles ont une interface différente.

Alors, écrivons un adaptateur :



Le code de près

la classe publique TurkeyAdapter implémente Duck {

Dinde dinde;

Tout d'abord, vous devez implémenter l'interface du type auquel vous vous adaptez. Il s'agit de l'interface que votre client s'attend à voir.

public TurkeyAdapter(Turquie dinde) {

cette.dinde = dinde;

}

Ensuite, nous devons obtenir une référence à l'objet que nous adaptions ; ici, nous le faisons via le constructeur.

public void charlatan() {

dinde.gobble();

}

Nous devons maintenant implémenter toutes les méthodes de l'interface ; la traduction quack() entre les classes est simple : il suffit d'appeler la méthode gobble().

public void voler() {

pour(int i=0; i < 5; i++) {

dinde.mouche();

}

}

}

Même si les deux interfaces ont une méthode fly(), les dindes volent par à-coups, elles ne peuvent pas voler sur de longues distances comme les canards. Pour établir une correspondance entre la méthode fly() d'un canard et celle d'une dinde, nous devons appeler la méthode fly() de la dinde cinq fois pour compenser.

Testez l'adaptateur

Il ne nous reste plus qu'à ajouter un peu de code pour tester notre adaptateur :

```
classe publique DuckTestDrive {
    public static void main(String[] args) {
        Canard canard = nouveau MallardDuck();

        Dinde dinde = new WildTurkey();
        Canard dindeAdapter = new TurkeyAdapter(dinde);

        System.out.println("La dinde dit...");
        turkey.gobble();
        dinde.mouche();

        System.out.println("\nLe canard dit...");
        testDuck(canard);

        System.out.println("\nLe TurkeyAdapter dit...");
        testDuck(turkeyAdapter);
    }

    static void testDuck(Canard canard) {
        canard.coin-coin();
        canard.mouche();
    }
}
```

Créons un canard...
... et une dinde.
Et puis enveloppez la dinde dans un TurkeyAdapter, ce qui la fait ressembler à un canard.

Alors, testons la dinde : faites-la engloutir, faites-la voler.

Testons maintenant le canard en appelant la méthode testDuck(), qui attend un objet Duck.

Maintenant le grand test : on essaie de faire passer la dinde pour un canard...

Voici notre méthode testDuck() ; elle récupère un canard et appelle ses méthodes quack() et fly().

Essai

Aide de la fenêtre d'édition de fichier Don'tForgetToDuck

```
% java DuckTestDrive
La Turquie dit...
Glougloutir
Je vole sur une courte distance

Le canard dit...
Charlatan
Je vole

Le TurkeyAdapter dit...
Gobble gobble
Je vole sur une courte distance Je
vole sur une courte distance Je
vole sur une courte distance Je
vole sur une courte distance Je
vole sur une courte distance
```

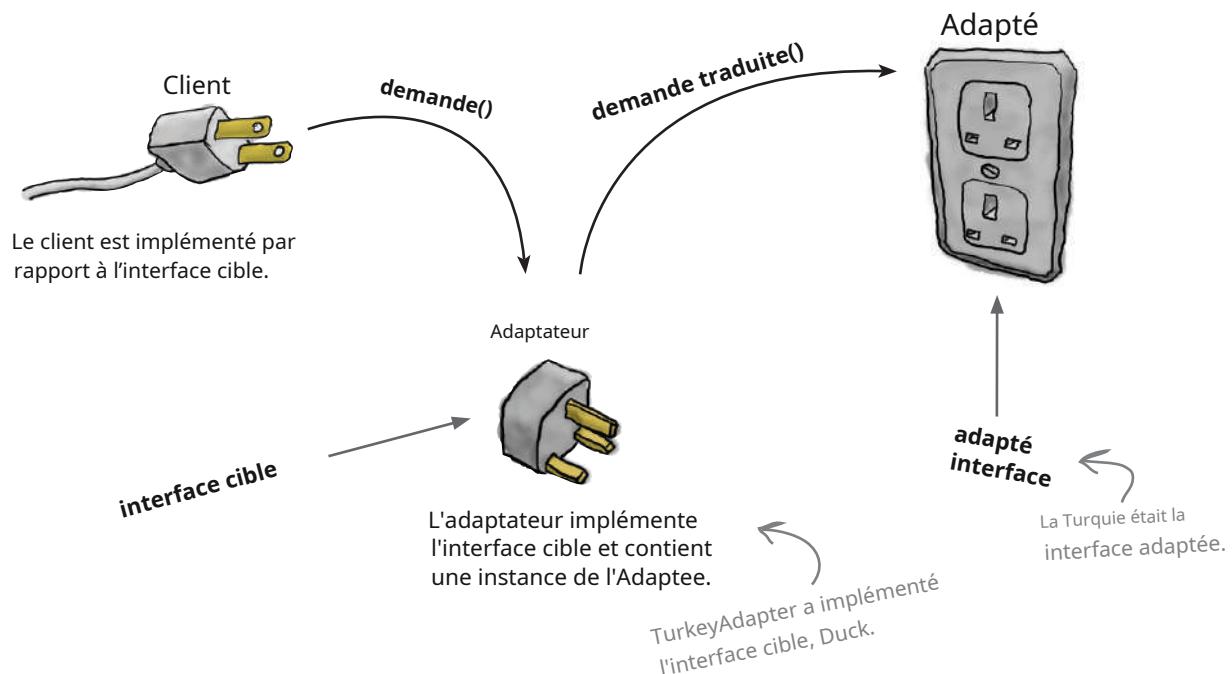
La dinde glougloute et vole sur une courte distance.

Le canard cancan et vole comme on peut s'y attendre.

Et l'adaptateur engloutit quand quack() est appelé et vole plusieurs fois quand fly() est appelé. La méthode testDuck() ne sait jamais qu'elle a une dinde déguisée en canard !

Le modèle d'adaptateur expliqué

Maintenant que nous avons une idée de ce qu'est un adaptateur, revenons en arrière et examinons à nouveau toutes les pièces.



Voici comment le client utilise l'adaptateur

- 1** Le client fait une demande à l'adaptateur en appelant une méthode sur celui-ci à l'aide de l'interface cible.
- 2** L'adaptateur traduit la demande en un ou plusieurs appels sur l'adapté à l'aide de l'interface de l'adapté.
- 3** Le client reçoit les résultats de l'appel et ne sait jamais qu'un adaptateur effectue la traduction.

Notez que le client et l'adapté sont découplés : aucun des deux n'a connaissance de l'existence de l'autre.



Disons que nous avons également besoin d'un adaptateur qui convertit un canard en dinde. Appelons-le DuckAdapter. Écrivez cette classe :

Comment avez-vous géré la méthode fly() (après tout, nous savons que les canards volent plus longtemps que les dindes) ? Consultez les réponses à la fin du chapitre pour connaître notre solution. Avez-vous pensé à une meilleure façon de procéder ?

there are no Dumb Questions

Q: Quelle est la quantité d'« adaptation » nécessaire à un adaptateur ? Il semble que si je dois implémenter une interface cible volumineuse, je pourrais avoir BEAUCOUP de travail sur les bras.

UN: Vous pourriez certainement le faire. La tâche d'implémentation d'un adaptateur est vraiment proportionnelle à la taille de l'interface que vous devez prendre en charge comme interface cible. Pensez cependant à vos options. Vous pourriez retravailler tous vos appels côté client à l'interface, ce qui entraînerait beaucoup de travail d'investigation et de modifications de code. Ou bien, vous pouvez fournir proprement une classe qui encapsule tous les changements dans cette classe.

Q: Un adaptateur encapsule-t-il toujours une et une seule classe ?

UN: Le rôle du modèle d'adaptateur est de convertir une interface en une autre. Bien que la plupart des exemples du modèle d'adaptateur montrent un adaptateur enveloppant un seul adapté, nous savons tous que le monde est souvent un peu plus compliqué. Ainsi, vous pouvez très bien avoir des situations où un adaptateur contient deux ou plusieurs adaptés qui sont nécessaires pour implémenter l'interface cible.

Cela fait référence à un autre motif appelé le motif de façade ; les gens confondent souvent les deux. Rappelez-nous de revenir sur ce point lorsque nous parlerons des façades plus loin dans ce chapitre.

Q: Que se passe-t-il si j'ai des parties anciennes et nouvelles de mon système, et que les parties anciennes attendent l'ancienne interface du fournisseur, mais que nous avons déjà écrit les nouvelles parties pour utiliser la nouvelle interface du fournisseur ? Cela va devenir déroutant d'utiliser un adaptateur ici et l'interface non encapsulée là. Ne serait-il pas préférable que j'écrive simplement mon ancien code et que j'oublie l'adaptateur ?

UN: Pas nécessairement. Vous pouvez par exemple créer un adaptateur bidirectionnel qui prend en charge les deux interfaces. Pour créer un adaptateur bidirectionnel, il vous suffit d'implémenter les deux interfaces concernées, afin que l'adaptateur puisse agir comme une ancienne interface ou une nouvelle interface.

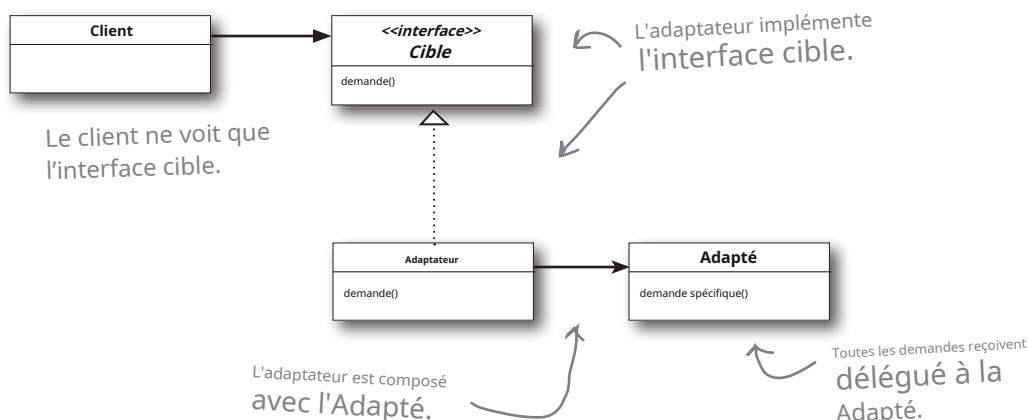
Modèle d'adaptateur défini

Assez de canards, de dindes et d'adaptateurs secteur ; soyons réalistes et examinons la définition officielle du modèle d'adaptateur :

Le modèle d'adaptateur convertit l'interface d'une classe en une autre interface attendue par les clients. L'adaptateur permet aux classes de fonctionner ensemble alors qu'elles ne pourraient pas le faire autrement en raison d'interfaces incompatibles.

Nous savons maintenant que ce modèle nous permet d'utiliser un client avec une interface incompatible en créant un adaptateur qui effectue la conversion. Cela permet de découpler le client de l'interface implémentée et, si nous prévoyons que l'interface change au fil du temps, l'adaptateur encapsule ce changement afin que le client n'ait pas à être modifié à chaque fois qu'il doit fonctionner avec une interface différente.

Nous avons examiné le comportement d'exécution du modèle ; examinons également son diagramme de classes :



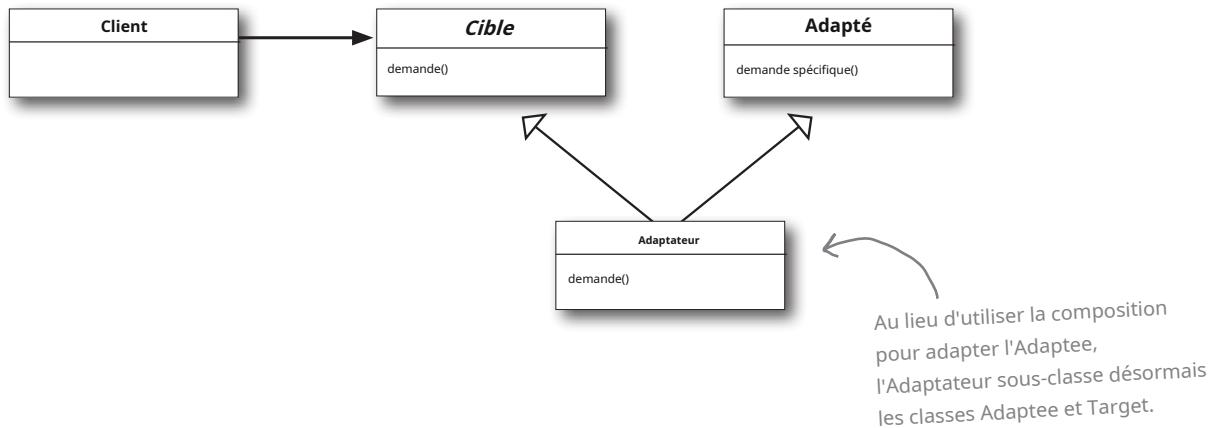
Le modèle Adapter est rempli de bons principes de conception orientée objet : découvrez l'utilisation de la composition d'objets pour envelopper l'adapté avec une interface modifiée. Cette approche a l'avantage supplémentaire de pouvoir utiliser un adaptateur avec n'importe quelle sous-classe de l'adapté.

Vérifiez également comment le modèle lie le client à une interface, et non à une implémentation ; nous pourrions utiliser plusieurs adaptateurs, chacun convertissant un ensemble différent de classes backend. Ou bien, nous pourrions ajouter de nouvelles implementations après coup, à condition qu'elles adhèrent à l'interface cible.

Adaptateurs d'objets et de classes

Maintenant, même si nous avons défini le modèle, nous ne vous avons pas encore raconté toute l'histoire. Il y a en fait *deux types* d'adaptateurs : *objet adaptateurs* et *classe adaptateurs*. Ce chapitre a abordé les adaptateurs d'objets, et le diagramme de classe de la page précédente est un diagramme d'un adaptateur d'objets.

Alors, qu'est-ce qu'un *classe adapter* et pourquoi ne vous en avons-nous pas parlé ? Parce que vous avez besoin de l'héritage multiple pour l'implémenter, ce qui n'est pas possible en Java. Mais cela ne signifie pas que vous ne rencontrerez peut-être pas le besoin d'adaptateurs de classe à l'avenir lorsque vous utiliserez votre langage d'héritage multiple préféré ! Regardons le diagramme de classes pour l'héritage multiple.



Cela vous semble familier ? C'est vrai, la seule différence est qu'avec un adaptateur de classe, nous sous-classons la cible et l'adapté, tandis qu'avec un adaptateur d'objet, nous utilisons la composition pour transmettre des requêtes à un adapté.



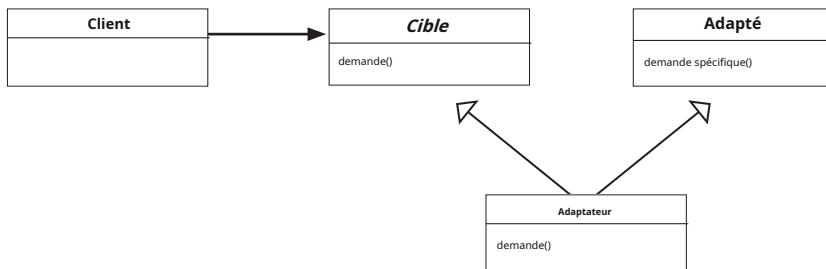
Les adaptateurs d'objet et les adaptateurs de classe utilisent deux moyens différents pour adapter l'adapté (composition ou héritage). Comment ces différences d'implémentation affectent-elles la flexibilité de l'adaptateur ?



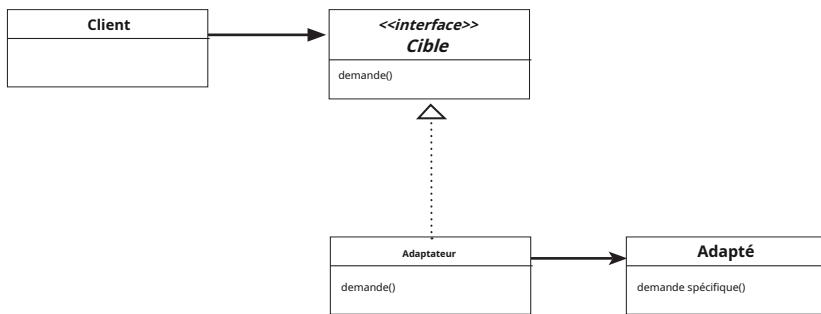
Aimants en forme de canard

Votre tâche consiste à prendre les aimants représentant le canard et la dinde et à les faire glisser sur la partie du diagramme qui décrit le rôle joué par cet oiseau, dans notre exemple précédent. (Essayez de ne pas revenir en arrière.) Ajoutez ensuite vos propres annotations pour décrire son fonctionnement.

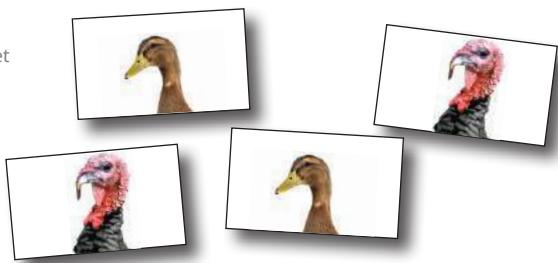
Adaptateur de classe



Adaptateur d'objet



Faites-les glisser sur le diagramme de classe pour montrer quelle partie du diagramme représente la classe Canard et laquelle représente la classe Dinde.

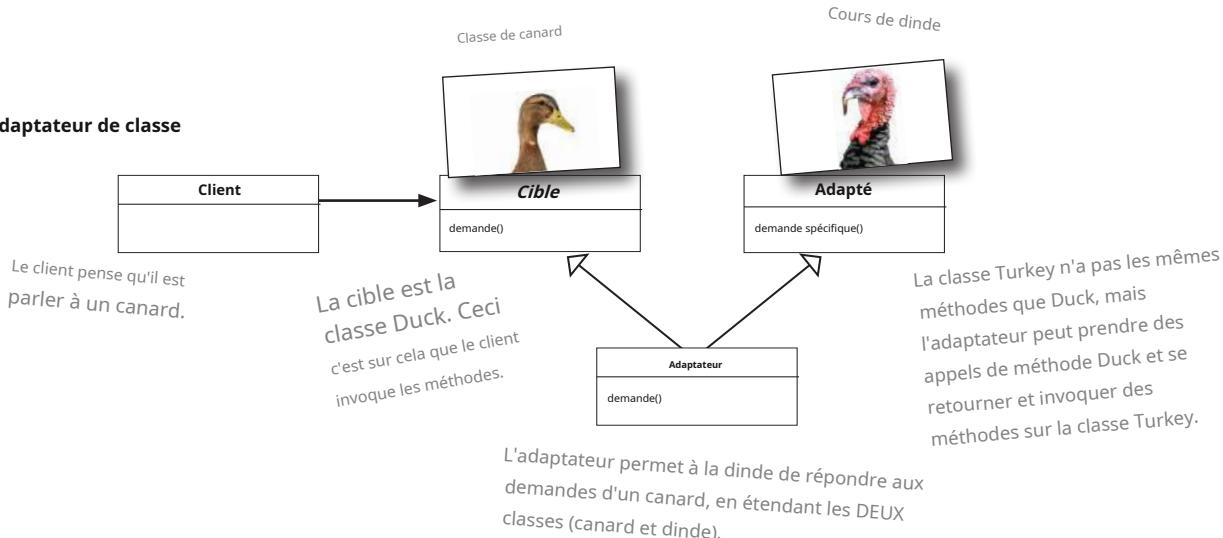




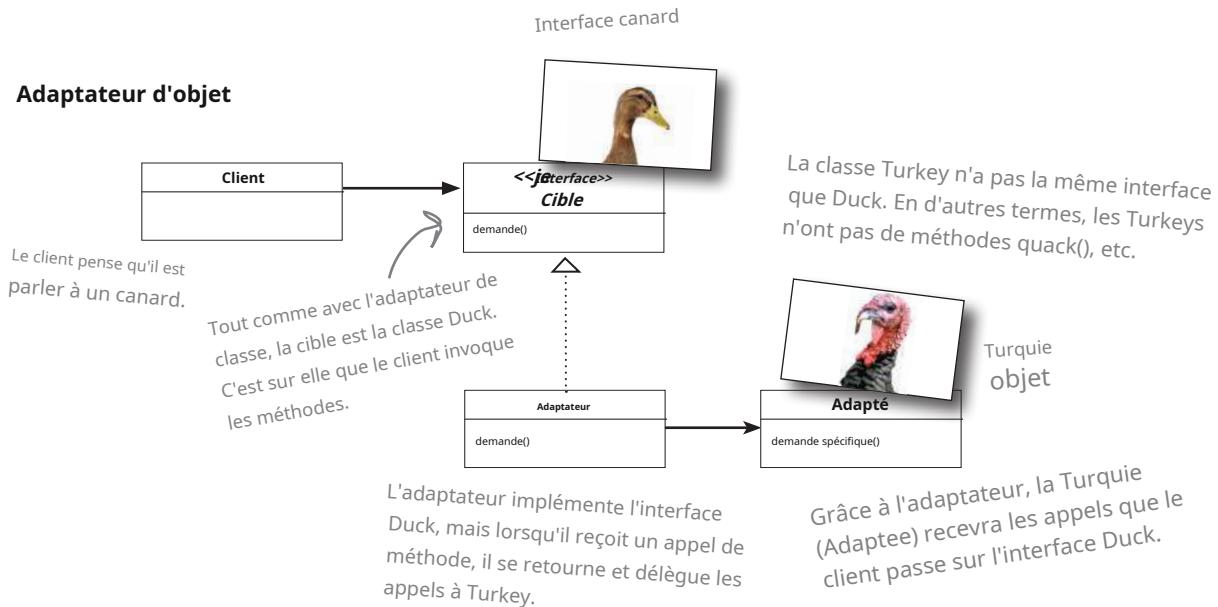
Aimants en forme de canard Répondre

Remarque : l'adaptateur de classe utilise l'héritage multiple, vous ne pouvez donc pas le faire en Java...

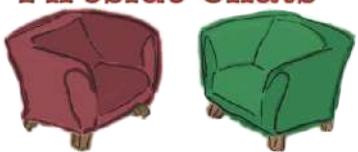
Adaptateur de classe



Adaptateur d'objet



Fireside Chats



La conférence de ce soir : **L'adaptateur d'objet et l'adaptateur de classe se rencontrent face à face.**

Adaptateur d'objet :

Comme j'utilise la composition, j'ai une longueur d'avance. Je peux adapter non seulement une classe adaptée, mais aussi n'importe laquelle de ses sous-classes.

Dans ma partie du monde, nous préférons utiliser la composition à l'héritage ; vous économiserez peut-être quelques lignes de code, mais tout ce que je fais, c'est écrire un peu de code à déléguer à l'adapté. Nous aimons garder les choses flexibles.

Vous vous inquiétez d'un petit objet ? Vous pouvez peut-être rapidement remplacer une méthode, mais tout comportement que j'ajoute à mon code d'adaptateur fonctionne avec ma classe d'adaptateur *et toutes* ses sous-classes.

Eh bien, allez, soyez indulgent, j'ai juste besoin de composer avec la sous-classe pour que cela fonctionne.

Tu veux voir du désordre ? Regarde-toi dans le miroir !

Adaptateur de classe :

C'est vrai, j'ai des problèmes avec ça parce que je suis attaché à une classe d'adapté spécifique, mais j'ai un énorme avantage parce que je n'ai pas à réimplémenter tout mon adapté. Je peux également remplacer le comportement de mon adapté si j'en ai besoin car je ne fais que sous-classer.

Flexible peut-être, mais efficace ? Non. Il n'y a qu'un seul moi, pas un adaptateur et un adapté.

Ouais, mais que se passe-t-il si une sous-classe d'Adaptee ajoute un nouveau comportement ? Et alors ?

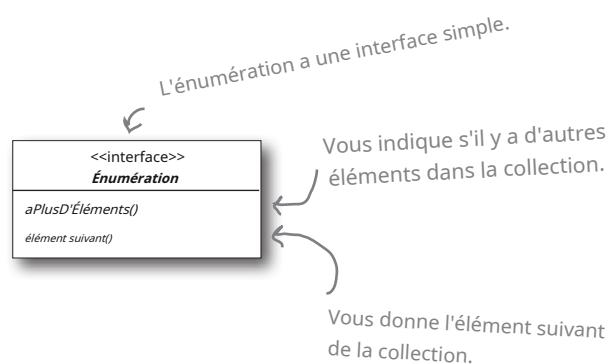
Ça a l'air compliqué...

Adaptateurs du monde réel

Jetons un œil à l'utilisation d'un simple adaptateur dans le monde réel (quelque chose de plus sérieux que les canards au moins)...

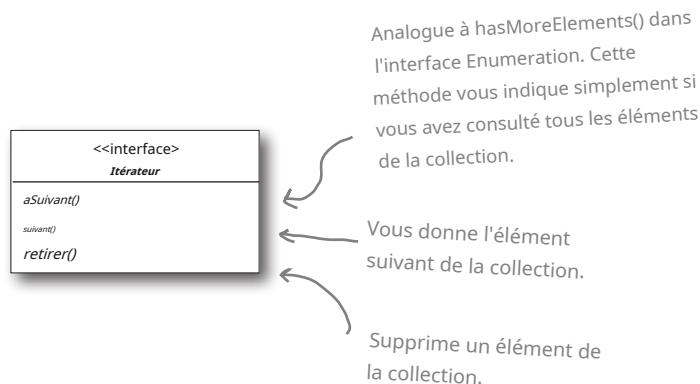
Recenseurs

Si vous utilisez Java depuis un certain temps, vous vous souvenez probablement que les premiers types de collection (Vector, Stack, Hashtable et quelques autres) implémentent une méthode, `elements()`, qui renvoie une énumération. L'interface Enumeration vous permet de parcourir les éléments d'une collection sans connaître les spécificités de leur gestion dans la collection.



Itérateurs

Les classes Collection les plus récentes utilisent une interface Iterator qui, comme l'interface Enumeration, vous permet de parcourir un ensemble d'éléments dans une collection et ajoute la possibilité de supprimer des éléments.

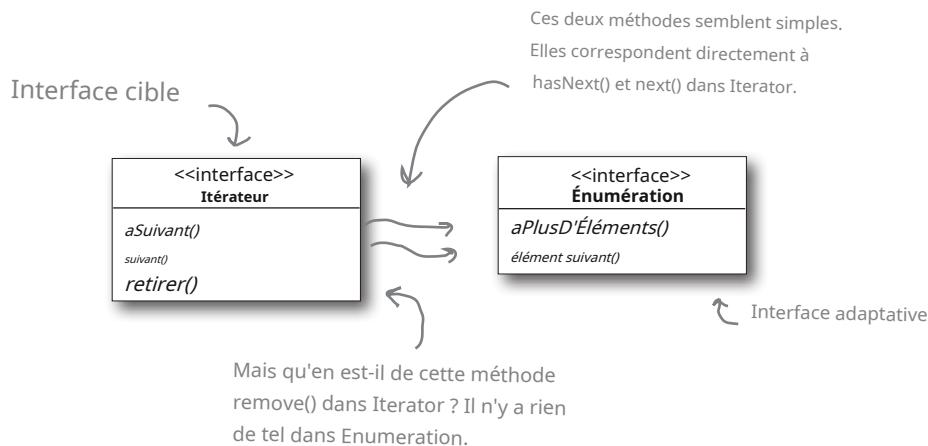


Utilisation d'énumérateurs avec du code qui attend des itérateurs

Nous sommes parfois confrontés à du code hérité qui expose l'interface Enumeration, mais nous aimerais que notre nouveau code utilise uniquement des Iterators. Il semble que nous devions créer un adaptateur.

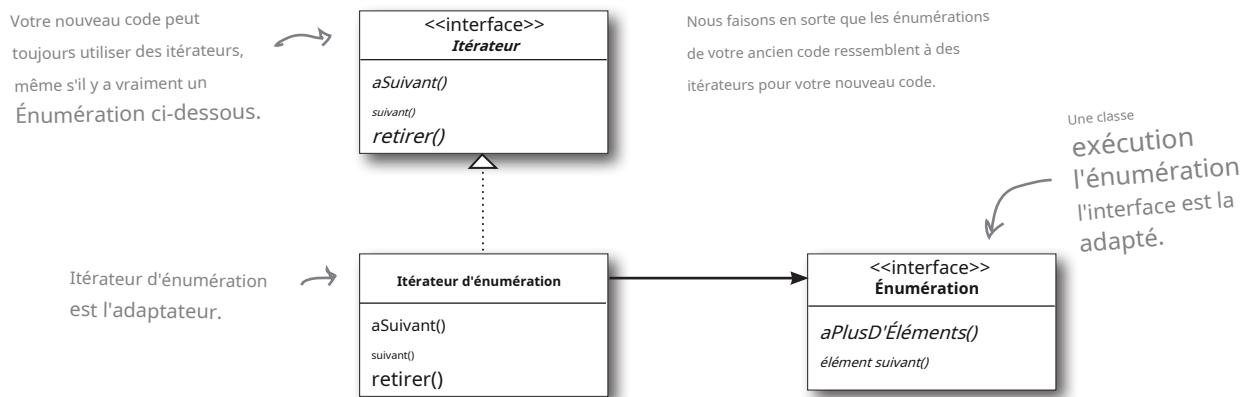
Adapter une énumération à un itérateur

Nous allons d'abord examiner les deux interfaces pour déterminer comment les méthodes sont mappées de l'une à l'autre. En d'autres termes, nous allons déterminer ce qu'il faut appeler sur l'adapté lorsque le client invoque une méthode sur la cible.



Conception de l'adaptateur

Voici à quoi devraient ressembler les classes : nous avons besoin d'un adaptateur qui implémente l'interface Target et qui est composé d'un adaptateur. Les méthodes `hasNext()` et `next()` vont être simples à mapper de la cible vers l'adaptateur : nous les passons simplement directement. Mais que faire de `remove()`? Pensez-y un instant (et nous y reviendrons sur la page suivante). Pour l'instant, voici le diagramme de classe :



Gestion de la méthode remove()

Eh bien, nous savons qu'Enumeration ne prend pas en charge remove(). C'est une interface en lecture seule. Il n'existe aucun moyen d'implémenter une méthode remove() entièrement fonctionnelle sur l'adaptateur. Le mieux que nous puissions faire est de lancer une exception d'exécution. Heureusement, les concepteurs de l'interface Iterator ont prévu ce besoin et ont défini la méthode remove() de manière à ce qu'elle prenne en charge une UnsupportedOperationException.

Il s'agit d'un cas où l'adaptateur n'est pas parfait ; les clients devront faire attention aux exceptions potentielles, mais tant que le client est prudent et que l'adaptateur est bien documenté, c'est une solution parfaitement raisonnable.

Écriture de l'adaptateur EnumerationIterator

Voici un code simple mais efficace pour toutes ces classes héritées qui produisent encore des énumérations :

```
classe publique EnumerationIterator implémente Iterator<Object> {
    Énumération<?> énumération;

    public EnumerationIterator(Enumeration<?> enumeration) {
        this.enumeration = énumération;
    }

    public boolean hasNext() {
        renvoyer l'énumération.hasMoreElements();
    }

    Objet public suivant() {
        renvoyer l'énumération.nextElement();
    }

    public void remove() {
        lancer une nouvelle UnsupportedOperationException();
    }
}
```

Puisque nous nous adaptons Énumération vers Iterator, notre adaptateur implémente l'interface Iterator... il doit ressembler à un Iterator.

L'énumération que nous sommes s'adapter. Nous utilisons composition, nous la stockons donc dans une variable d'instance.

La méthode hasNext() de l'Itérateur est déléguée à la méthode hasMoreElements() de l'Énumération...

... et la méthode next() de l'Itérateur est déléguée à la méthode nextElement() de l'Énumération.

Malheureusement, nous ne pouvons pas prendre en charge la méthode remove() d'Iterator, nous devons donc abandonner (en d'autres termes, nous abandonnons !). Ici, nous lançons simplement une exception.



Bien que Java soit allé dans le sens de l'interface Iterator, il existe néanmoins toujours du code client hérité qui dépend de l'interface Enumeration, donc un adaptateur qui convertit un Iterator en Enumeration pourrait potentiellement être utile.

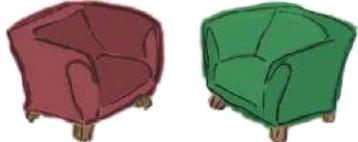
Écrivez un adaptateur qui adapte un itérateur à une énumération. Vous pouvez tester votre code en adaptant une ArrayList. La classe ArrayList prend en charge l'interface Iterator mais ne prend pas en charge les énumérations.



Certains adaptateurs secteur font plus que simplement changer l'interface—ils ajoutent d'autres fonctionnalités comme une protection contre les surtensions, des voyants lumineux et d'autres gadgets.

Si vous deviez implémenter ce type de fonctionnalités, quel modèle utiliseriez-vous ?

Fireside Chats



La conférence de ce soir :**Le modèle Décorateur et le modèle Adaptateur discutent de leurs différences.**

Décorateur:

Je suis important. Mon travail consiste à *responsabilité*—vous savez que lorsqu'un décorateur est impliqué, de nouvelles responsabilités ou de nouveaux comportements seront ajoutés à votre conception.

C'est peut-être vrai, mais ne pensez pas que nous ne travaillons pas dur. Lorsque nous devons décorer une grande interface, ouah, cela peut demander beaucoup de code.

C'est mignon. Ne pensez pas que nous recevons toute la gloire ; parfois, je ne suis qu'un décorateur qui se fait embobiner par je ne sais combien d'autres décorateurs. Lorsqu'un appel de méthode vous est délégué, vous n'avez aucune idée du nombre d'autres décorateurs qui l'ont déjà traité et vous ne savez pas si vous serez un jour remarqué pour vos efforts dans le traitement de la demande.

Adaptateur:

Vous, les décorateurs, voulez toute la gloire, tandis que nous, les adaptateurs, sommes dans les tranchées et faisons le sale boulot : convertir les interfaces. Nos missions ne sont peut-être pas très glamour, mais nos clients apprécient que nous leur simplifions la vie.

Essayez de jouer les adaptateurs lorsque vous devez réunir plusieurs classes pour fournir l'interface que votre client attend. C'est difficile. Mais nous avons un dicton : « Un client découplé est un client heureux. »

Eh bien, si les adaptateurs font leur travail, nos clients ne savent même pas que nous sommes là. Cela peut être un travail ingrat.

Décorateur:

Eh bien, nous, les décorateurs, le faisons aussi, mais nous sommes les seuls à le permettre. *nouveau comportement* à ajouter aux classes sans modifier le code existant. Je continue à dire que les adaptateurs ne sont que des décorateurs sophistiqués. Je veux dire, tout comme nous, vous enveloppez un objet.

Euh, non. Notre travail dans la vie est d'étendre les comportements ou les responsabilités des objets que nous emballons ; nous ne sommes pas un *passage simple*.

Peut-être devrions-nous accepter d'être en désaccord. Nous semblons nous ressembler sur le papier, mais il est clair que nous sommes *millésâ* part dans notre *intention*.

Adaptateur:

Mais l'avantage de nos adaptateurs est que nous permettons aux clients d'utiliser de nouvelles bibliothèques et de nouveaux sous-ensembles sans changer *n'importe lequel* code ; ils comptent simplement sur nous pour effectuer la conversion à leur place. Hé, c'est un créneau, mais nous sommes bons dans ce domaine.

Non, non, non, pas du tout. Nous *toujours* convertir l'interface de ce que nous emballons ; vous *jamais* faire. Je dirais qu'un décorateur est comme un adaptateur ; c'est juste que vous ne changez pas l'interface !

Hé, qui appelles-tu un simple passage ? Viens et on verra combien de temps ça va durer *toi* dernière conversion de quelques interfaces !

Oh oui, je suis avec toi là-dessus.

Et maintenant quelque chose de différent...

Il y a un autre modèle dans ce chapitre.

Vous avez vu comment le modèle d'adaptateur convertit l'interface d'une classe en une interface attendue par un client. Vous savez également que nous y parvenons en Java en encapsulant l'objet qui a une interface incompatible avec un objet qui implémente la bonne.

Nous allons maintenant examiner un modèle qui modifie une interface, mais pour une raison différente : simplifier l'interface. Il est bien nommé modèle Facade car ce modèle cache toute la complexité d'une ou plusieurs classes derrière une façade propre et bien éclairée.



Associez chaque modèle à son intention :

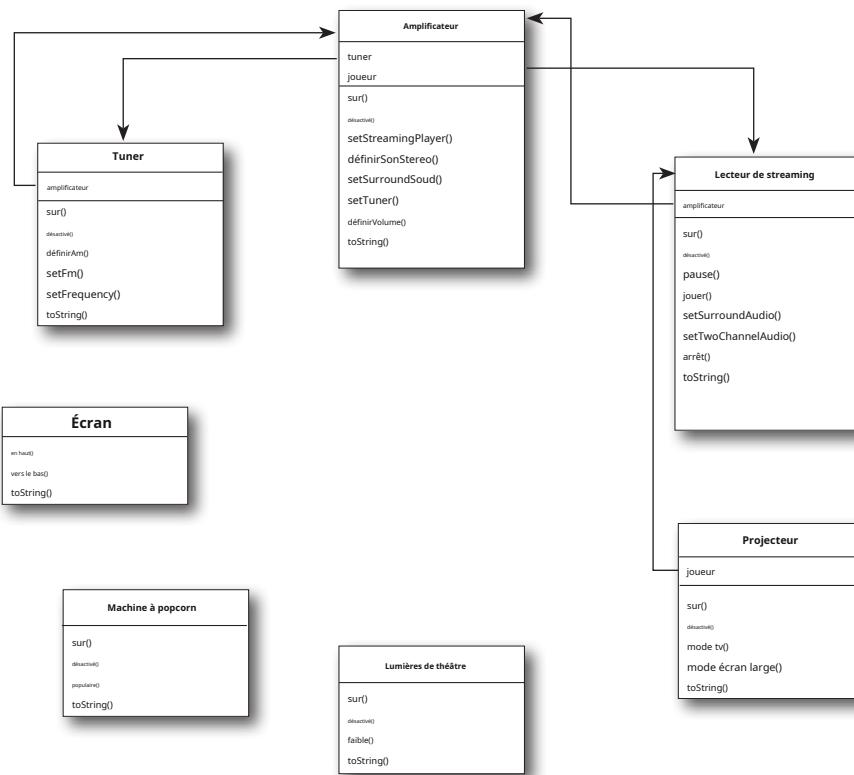
Modèle	Intention
Décorateur	Convertit une interface en une autre
Adaptateur	Ne modifie pas l'interface, mais ajoute de la responsabilité
Façade	Rend une interface plus simple

Home Sweet Home Cinéma

Avant de plonger dans les détails du modèle de façade, examinons une obsession nationale croissante : construire un joli cinéma pour regarder en boucle tous ces films et séries télévisées.

Vous avez fait vos recherches et vous avez assemblé un système complet comprenant un lecteur de streaming, un système de projection vidéo, un écran automatisé, un son surround et même un appareil à pop-corn.

Regardez tous les composants que vous avez assemblés :



C'est beaucoup de cours, beaucoup des interactions, et un grand ensemble des interfaces à apprendre et utiliser.

Vous avez passé des semaines à installer des câbles, à monter le projecteur, à effectuer toutes les connexions et à peaufiner le tout. Il est maintenant temps de mettre tout cela en marche et de profiter d'un film...

Regarder un film (à la dure)

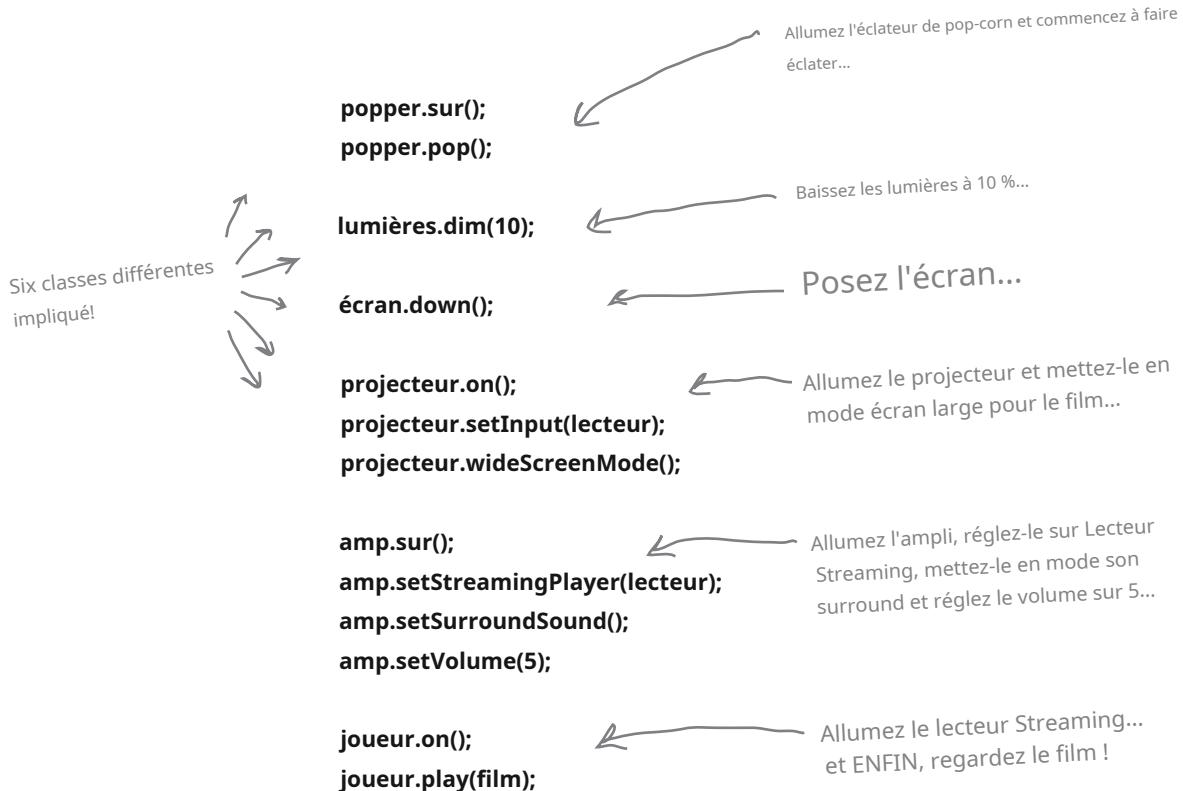
Choisissez un film, détendez-vous et préparez-vous à la magie du cinéma. Oh, il n'y a qu'une chose : pour regarder le film, vous devez effectuer quelques tâches :

- ❶ Allumez l'éclateur de pop-corn
- ❷ Commencez à faire éclater le popper.
- ❸ Tamisez les lumières.
- ❹ Abaissez l'écran
- ❺ Allumez le projecteur
- ❻ Régler l'entrée du projecteur sur le lecteur de streaming
- ❼ Mettez le projecteur en mode écran large
- ❽ Allumez l'amplificateur de son
- ❾ Réglez l'amplificateur sur l'entrée du lecteur de streaming
- ❿ Réglez l'amplificateur sur le son surround Réglez
- ⓫ le volume de l'amplificateur sur moyen (5)
- ⓬ Allumez le lecteur de streaming
- ⓭ Commencer à lire le film

Je suis déjà épuisé
et tout ce que j'ai fait c'est tourner
tout est en marche !



Examinons ces mêmes tâches en termes de classes et d'appels de méthodes nécessaires pour les exécuter :



Mais il y a plus...

- ƒ Quand le film est terminé, comment éteindre tout ? Ne faudrait-il pas tout recommencer, à l'envers ?
- ƒ Ne serait-il pas aussi complexe d'écouter la radio ?
- ƒ Si vous décidez de mettre à niveau votre système, vous devrez probablement apprendre une procédure légèrement différente.

Alors que faire ? La complexité de l'utilisation de votre home cinéma devient évidente ! Voyons comment le motif de façade peut nous sortir de ce pétrin pour que nous puissions profiter du film...

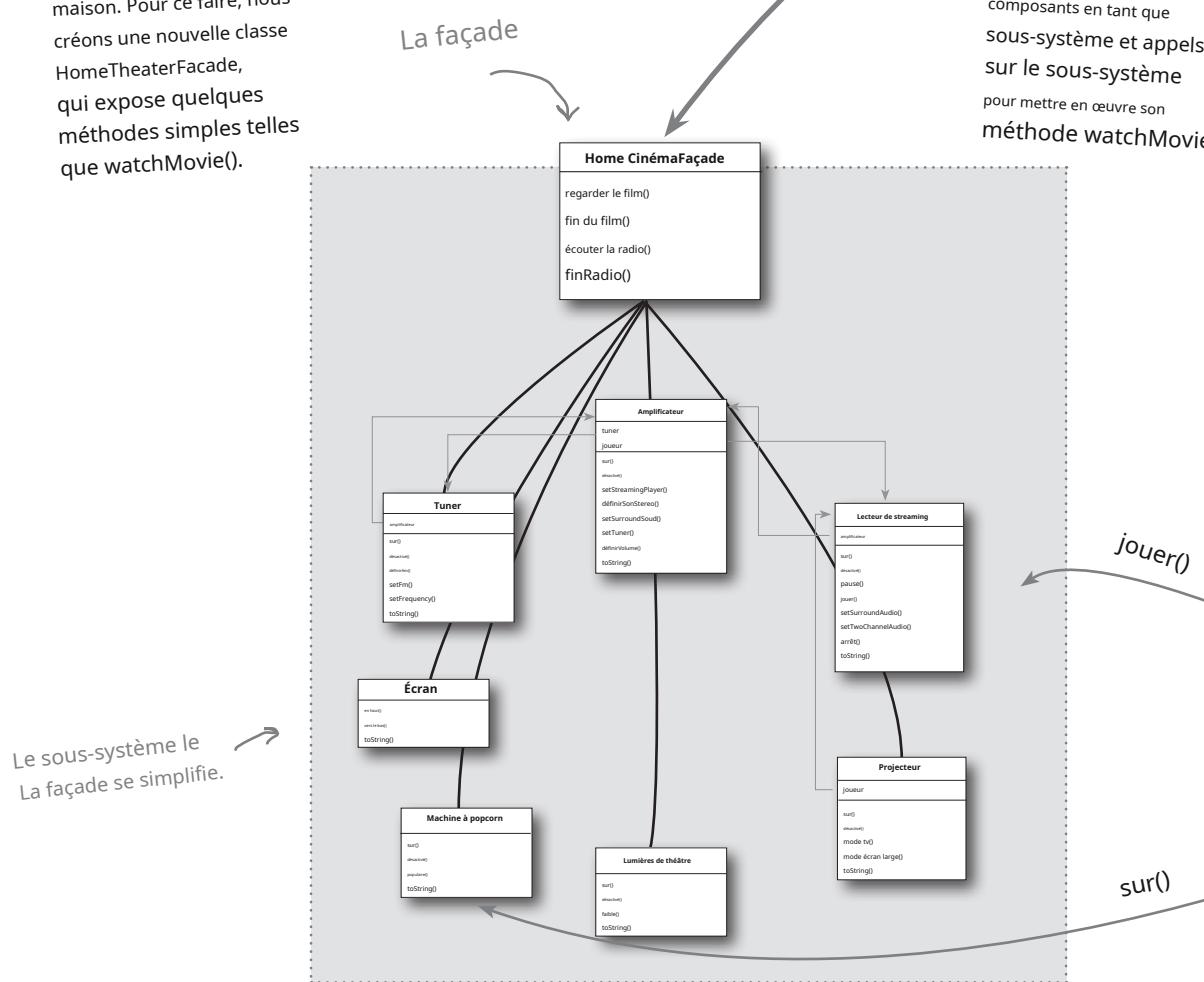
Lumières, caméra, façade !

Une façade est exactement ce dont vous avez besoin : avec le modèle Facade, vous pouvez prendre un sous-système complexe et le rendre plus facile à utiliser en implémentant une classe Facade qui fournit une interface plus raisonnable. Ne vous inquiétez pas ; si vous avez besoin de la puissance du sous-système complexe, il est toujours là pour vous, mais si tout ce dont vous avez besoin est une interface simple, la façade est là pour vous.

Voyons comment fonctionne la façade :

- 1 Ok, il est temps de créer une façade pour le système de cinéma maison. Pour ce faire, nous créons une nouvelle classe HomeTheaterFacade, qui expose quelques méthodes simples telles que watchMovie().

- 2 La classe Façade traite du home cinéma composants en tant que sous-système et appels sur le sous-système pour mettre en œuvre son méthode watchMovie().





- ➊ Votre code client appelle désormais des méthodes sur la façade du home cinéma, et non sur le sous-système. Ainsi, pour regarder un film, il suffit d'appeler une méthode, `watchMovie()`, et elle communique avec les lumières, le lecteur de streaming, le projecteur, l'amplificateur, l'écran et la machine à pop-corn pour nous.



- ➋ La façade laisse toujours le sous-système accessible afin qu'il puisse être utilisé directement. Si vous avez besoin des fonctionnalités avancées des classes du sous-système, elles sont à votre disposition.

there are no Dumb Questions

Q:

Si la façade encapsule les classes du sous-système, comment un client ayant besoin de fonctionnalités de niveau inférieur peut-il y accéder ?

UN:

Les façades n'encapsulent pas les classes du sous-système ; elles fournissent simplement une interface simplifiée à leurs fonctionnalités. Les classes du sous-système restent disponibles pour une utilisation directe par les clients qui ont besoin d'utiliser des interfaces plus spécifiques. C'est une propriété intéressante du modèle Facade : il fournit une interface simplifiée tout en exposant toutes les fonctionnalités du système à ceux qui peuvent en avoir besoin.

Q:

La façade ajoute-t-elle des fonctionnalités ou transmet-elle simplement chaque requête au sous-système ?

UN:

Une façade est libre d'ajouter ses propres « intelligences » en plus d'utiliser le sous-système. Par exemple, bien que notre façade de cinéma maison n'implémente aucun nouveau comportement, elle est suffisamment intelligente pour savoir que l'électeur de popcorn doit être allumé avant qu'il ne puisse éclater (ainsi que les détails sur la façon d'allumer et d'organiser une projection de film).

Q:

Chaque sous-système n'a-t-il qu'une seule façade ?

UN:

Pas nécessairement. Le modèle permet certainement de créer n'importe quel nombre de façades pour un sous-système donné.

Q:

Quel est l'avantage de la façade autre que le fait que j'ai maintenant une interface plus simple ?

UN:

Le modèle de façade vous permet également de découpler l'implémentation de votre client de n'importe quel sous-système. Supposons que vous obteniez une augmentation importante et que vous décidiez de mettre à niveau votre home cinéma avec tous les nouveaux composants dotés d'interfaces différentes. Eh bien, si vous avez codé votre client sur la façade plutôt que sur le sous-système, votre code client n'a pas besoin de changer, juste la façade (et avec un peu de chance, le fabricant la fournit !).

Q:

Donc, la différence entre le modèle d'adaptateur et le modèle de façade est que l'adaptateur enveloppe une classe et la façade peut représenter plusieurs classes ?

UN:

Non ! N'oubliez pas que le modèle d'adaptateur transforme l'interface d'une ou plusieurs classes en une interface que le client attend. Bien que la plupart des exemples de manuels montrent que l'adaptateur adapte une classe, vous devrez peut-être adapter de nombreuses classes pour fournir l'interface pour laquelle un client est codé. De même, une façade peut fournir une interface simplifiée à une seule classe avec une interface très complexe.

La différence entre les deux ne réside pas dans le nombre de classes qu'ils « encapsulent », mais dans leur objectif. L'objectif du modèle d'adaptateur est de modifier une interface afin qu'elle corresponde à celle attendue par un client. L'objectif du modèle de façade est de fournir une interface simplifiée à un sous-système.

Une façade non simplifie seulement une interface, il découpe un client à partir d'un sous-système des composants.

Façades et les adaptateurs peuvent envelopper plusieurs cours, mais un l'intention de la façade est pour simplifier, tandis que un adaptateur est de convertir l'interface à quelque chose différent.

Construire la façade de votre home cinéma

Passons en revue la construction de la classe HomeTheaterFacade. La première étape consiste à utiliser la composition pour que la façade ait accès à tous les composants du sous-système :

```
classe publique HomeTheaterFacade {
```

```
    Amplificateur amp;
```

```
    Accordeur accordeur;
```

```
    Lecteur StreamingPlayer ;
```

```
    Projecteur projecteur;
```

```
    Lumières de TheaterLights ;
```

```
    Écran écran;
```

```
    Éclateur de popcorn PopcornPopper;
```

Voici la composition ; ce sont tous les composants du sous-système que nous allons utiliser.



```
public HomeTheaterFacade(Amplificateur amp,
```

```
    Accordeur accordeur,
```

```
    Lecteur StreamingPlayer ;
```

```
    Projecteur projecteur,
```

```
    Écran d'écran,
```

```
    Lumières de TheaterLights,
```

```
    (pop-corn) {
```

La façade reçoit une référence à chaque composant du sous-système dans son constructeur. La façade affecte ensuite chacun à la variable d'instance correspondante.



```
        ceci.amp = amp;
```

```
        ceci.tuner = tuner;
```

```
        this.player = joueur;
```

```
        this.projector = projecteur;
```

```
        this.screen = écran;
```

```
        ceci.lumières = lumières;
```

```
        ceci.popper = popper;
```

```
}
```

```
// autres méthodes ici
```



Nous sommes sur le point de les remplir...

```
}
```

Mise en œuvre de l'interface simplifiée

Il est maintenant temps de rassembler les composants du sous-système dans une interface unifiée.

Implémentons les méthodes watchMovie() et endMovie() :

```
public void watchMovie(Chaîne de film) {  
    System.out.println("Préparez-vous à regarder un film...");  
    popper.on();  
    popper.pop();  
    lumières.dim(10);  
    écran.down();  
    projecteur.on();  
    projecteur.wideScreenMode();  
    amp.on();  
    amp.setStreamingPlayer(lecteur);  
    amp.setSurroundSound();  
    amp.setVolume(5);  
    joueur.on();  
    joueur.play(film);  
}
```

```
public void endMovie() {  
    System.out.println("Fermeture du cinéma..."); popper.off();  
  
    lumières.allumées();  
    écran.up();  
    projecteur.off();  
    amp.off();  
    joueur.stop();  
    joueur.off();  
}
```

watchMovie() suit la même séquence que nous devions faire à la main auparavant, mais l'encapsule dans une méthode pratique qui fait tout le travail. Notez que pour chaque tâche, nous déléguons la responsabilité au composant correspondant dans le sous-système.

Et endMovie() se charge de tout arrêter pour nous. Encore une fois, chaque tâche est déléguée au composant approprié du sous-système.



Pensez aux façades que vous avez rencontrées dans l'API Java.
Où aimeriez-vous en avoir quelques-unes nouvelles ?



Il est temps de regarder un film (de manière simple)

C'est l'heure du spectacle !

```
classe publique HomeTheaterTestDrive {
    public static void main(String[] args) {
        // instancier les composants ici
```

Ici, nous créons les composants directement lors de la phase de test. Normalement, le client reçoit une façade ; il n'a pas besoin d'en construire une lui-même.

```
HomeTheaterFacade homeTheater =
    nouveau HomeTheaterFacade(ampli, tuner, joueur,
    projecteur, écran, lumières, popper);
```

Tout d'abord, vous instanciez la Façade avec tous les composants du sous-système.

```
homeTheater.watchMovie("Les Aventuriers de l'Arche perdue");
homeTheater.endMovie();
}
```

Utilisez l'interface simplifiée pour démarrer d'abord le film, puis l'arrêter.

}

Voici le résultat.

Appeler les façades watchMovie() fait tout ce travail pour nous..

```
%java HomeTheaterTestDrive
Préparez-vous à regarder un film...
Popcorn Popper sur
Popcorn Popper fait éclater du popcorn ! Les lumières
du plafond du théâtre s'atténuent à 10 % L'écran du
théâtre s'abaisse
Projecteur allumé
Projecteur en mode écran large (format d'image 16x9)
Amplificateur allumé
Réglage de l'amplificateur Lecteur de streaming sur Lecteur de streaming Son
surround de l'amplificateur activé (5 haut-parleurs, 1 caisson de basses) Réglage du
volume de l'amplificateur sur 5
Lecteur de streaming sur
Lecteur de streaming diffusant « Les Aventuriers de l'Arche
perdue » Fermeture du cinéma...
Éclateur de pop-corn éteint
Les lumières du plafond du
théâtre s'allument, le
projecteur est éteint
Amplificateur éteint
Streaming Player arrêté "Les Aventuriers de l'Arche perdue"
Streaming Player désactivé
%
```

... et là, nous avons fini de regarder le film, donc appeler endMovie() désactive tout.

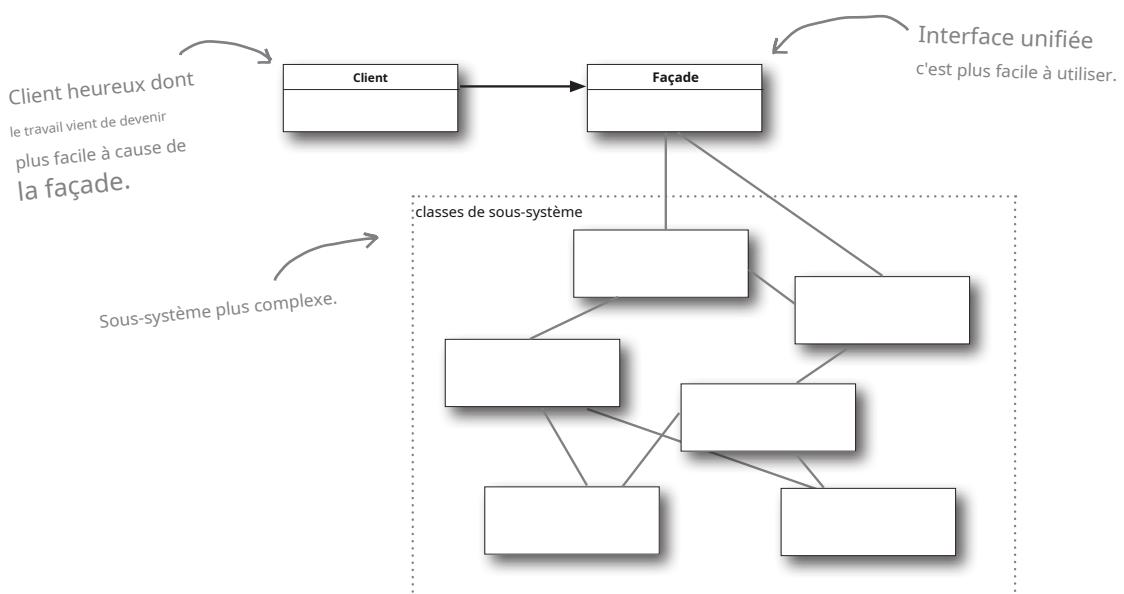
Motif de façade défini

Pour utiliser le modèle Facade, nous créons une classe qui simplifie et unifie un ensemble de classes plus complexes appartenant à un sous-système. Contrairement à de nombreux modèles, Facade est assez simple ; il n'y a pas d'abstractions complexes à comprendre. Mais cela ne le rend pas moins puissant : le modèle Facade nous permet d'éviter un couplage étroit entre les clients et les sous-systèmes et, comme vous le verrez bientôt, nous aide également à adhérer à un nouveau principe orienté objet.

Avant de présenter ce nouveau principe, examinons la définition officielle du modèle :

Le motif de la façadefournit une interface unifiée à un ensemble d'interfaces dans un sous-système. Façade définit une interface de niveau supérieur qui rend le sous-système plus facile à utiliser.

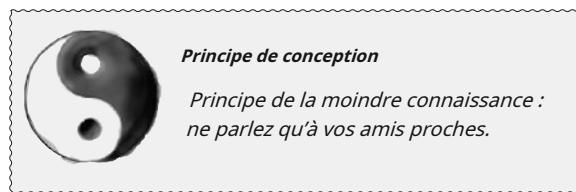
Il n'y a pas grand-chose ici que vous ne sachiez déjà, mais l'une des choses les plus importantes à retenir à propos d'un modèle est son intention. Cette définition nous dit haut et fort que le but de la façade est de rendre un sous-système plus facile à utiliser grâce à une interface simplifiée. Vous pouvez le voir dans le diagramme de classes du modèle :



Et voilà, vous avez un nouveau modèle à votre actif ! Il est maintenant temps d'appliquer ce nouveau principe OO. Attention, celui-ci peut remettre en cause certaines hypothèses !

Le principe de la moindre connaissance

Le principe de la moindre connaissance nous pousse à réduire les interactions entre objets à quelques « amis » proches. Le principe est généralement formulé ainsi :



Mais qu'est-ce que cela signifie concrètement ? Cela signifie que lorsque vous concevez un système, quel que soit l'objet, vous devez faire attention au nombre de classes avec lesquelles il interagit et également à la manière dont il interagit avec ces classes.

Ce principe nous empêche de créer des conceptions comportant un grand nombre de classes couplées entre elles, de sorte que les modifications apportées à une partie du système se répercutent sur d'autres parties. Lorsque vous créez de nombreuses dépendances entre de nombreuses classes, vous créez un système fragile qui sera coûteux à maintenir et complexe à comprendre pour les autres.



À combien de classes ce code est-il couplé ?

```
flotteur public getTemp() {  
    station de retour.getThermometer().getTemperature();  
}
```

Comment NE PAS se faire des amis et influencer les objets

D'accord, mais comment éviter cela ? Le principe fournit quelques lignes directrices : prenez n'importe quel objet et, à partir de n'importe quelle méthode de cet objet, appelez uniquement les méthodes qui appartiennent à :

- f L'objet lui-même
- f Objets passés en paramètre à la méthode
- f Tout objet que la méthode crée ou instancie
- f Tous les composants de l'objet



Notez que ces directives nous indiquent de ne pas appeler de méthodes sur des objets qui ont été renvoyés après l'appel d'autres méthodes !!

Considérez un « composant » comme tout objet référencé par une variable d'instance. En d'autres termes, considérez cela comme une relation HAS-A.

Cela semble un peu strict, n'est-ce pas ? Quel mal y a-t-il à appeler la méthode d'un objet que nous obtenons en retour d'un autre appel ? Eh bien, si nous devions faire cela, nous ferions alors une requête sur la sous-partie d'un autre objet (et augmenterions le nombre d'objets que nous connaissons directement). Dans de tels cas, le principe nous oblige à demander à l'objet de faire la requête pour nous ; de cette façon, nous n'avons pas besoin de connaître les objets qui le composent (et nous gardons notre cercle d'amis restreint). Par exemple :

Sans le
Principe

```
flotteur public getTemp() {
    Thermomètre thermomètre = station.getThermometer();
    return thermomètre.getTemperature();
}
```



Ici, nous récupérons l'objet thermomètre de la station, puis appelons nous-mêmes la méthode getTemperature().

Avec le
Principe

```
flotteur public getTemp() {
    station de retour.getTemperature();
}
```



Lorsque nous appliquons le principe, nous ajoutons une méthode à la classe Station qui effectue la requête au thermomètre à notre place. Cela réduit le nombre de classes dont nous dépendons.

Gardez vos appels de méthode dans les limites...

Voici une classe Car qui montre toutes les manières dont vous pouvez appeler des méthodes tout en respectant le principe de moindre connaissance :

```
classe publique Voiture {
    Moteur moteur;
    // autres variables d'instance
```

Voici un composant de cette classe. Nous pouvons appeler ses méthodes.

```
voiture publique() {
    // initialiser le moteur, etc.
}
```

Ici, nous créons un nouvel objet ; ses méthodes sont légales.

```
public void start(Clé clé) {
    Portes portes = nouvelles Portes();
    booléen autorisé =
        clé.tours();
    si (autorisé) {
        moteur.start();
        updateDashboardDisplay();
        portes.lock();
    }
}
```

Vous pouvez appeler une méthode sur un objet passé en paramètre.

Vous pouvez appeler une méthode sur un composant de l'objet.

Vous pouvez appeler une méthode locale dans l'objet.

Vous pouvez appeler une méthode sur un objet que vous créez ou instanciez.

```
public void updateDashboardDisplay() {
    // mettre à jour l'affichage
}
```

}

*there are no
Dumb Questions*

Q:

Il existe un autre principe appelé la loi de Déméter ; quel est leur lien ?

UN:

Les deux termes sont identiques et vous les verrez utilisés de manière interchangeable. Nous préférerons utiliser le principe de moindre connaissance pour plusieurs raisons : (1) le nom est plus intuitif et (2) l'utilisation du mot « loi » implique que nous devons toujours appliquer ce principe. En fait, aucun principe n'est une loi ; tous les principes devraient

être utilisés quand et où ils sont utiles. Toute conception implique des compromis (abstractions contre vitesse, espace contre temps, etc.) et bien que les principes fournissent des orientations, vous devez prendre en compte tous les facteurs avant de les appliquer.

Q:

Y a-t-il des inconvénients à appliquer le principe de la moindre connaissance ?

UN:

Qui même si le principe réduit les dépendances entre les objets et que des études ont montré que cela réduit la maintenance logicielle, il est également vrai que l'application de ce principe entraîne l'écriture de davantage de classes « wrapper » pour gérer les appels de méthodes vers d'autres composants. Cela peut entraîner une augmentation de la complexité et du temps de développement ainsi qu'une diminution des performances d'exécution.

Sharpen your pencil

Maison publique {
 Station Météo;

// autres méthodes et constructeur

```
flotteur public getTemp() {  
    station de retour.getThermometer().getTemperature();  
}  
}
```

Maison publique {
 Station Météo;

// autres méthodes et constructeur

```
flotteur public getTemp() {  
    Thermomètre thermomètre = station.getThermometer(); return  
    getTempHelper(thermomètre);  
}  
  
public float getTempHelper(Thermomètre thermomètre) {  
    retourner thermomètre.getTemperature();  
}  
}
```



Zone de casque de sécurité.
attention à
hypothèses en baisse

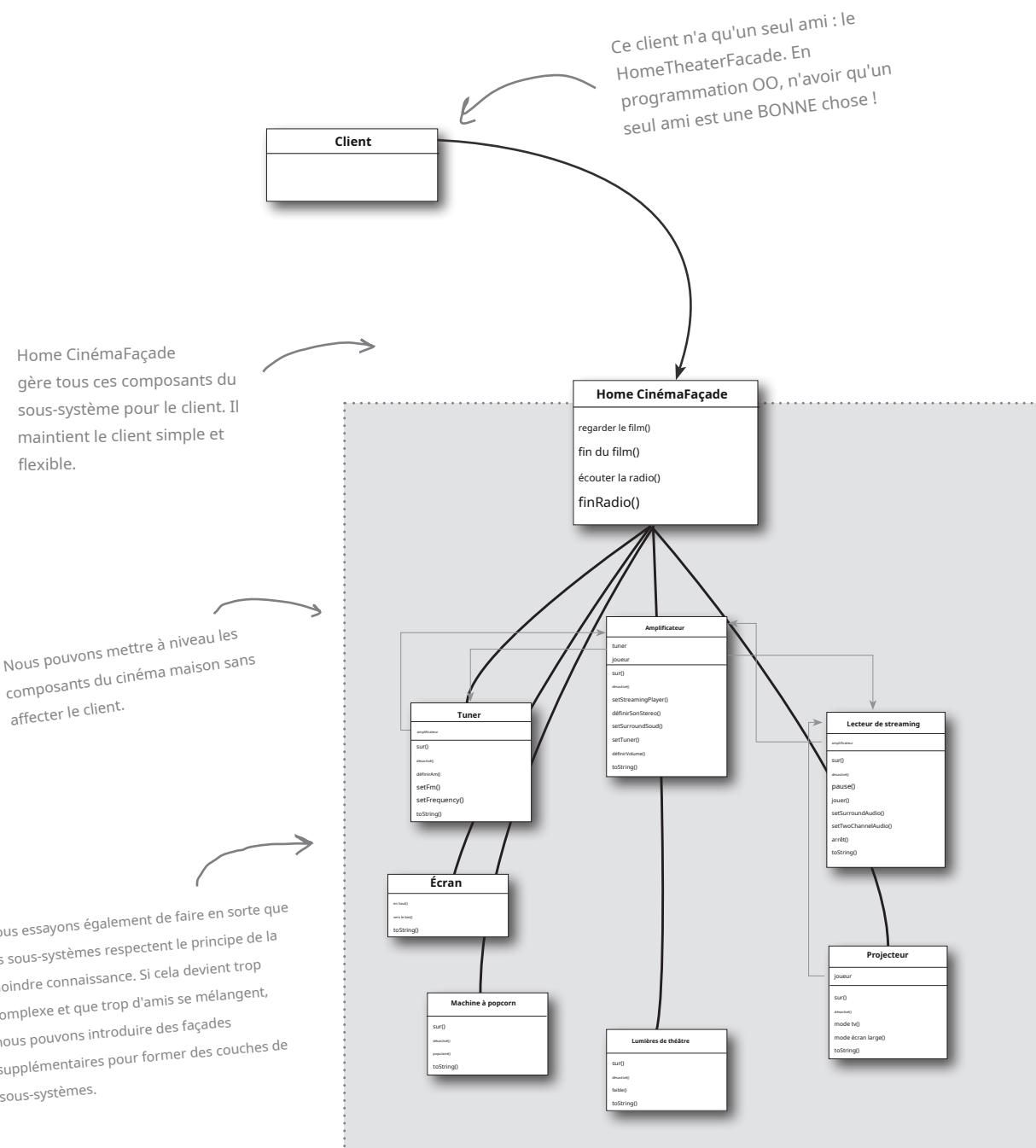


Pouvez-vous penser à une utilisation courante de Java
qui viole le principe de moindre connaissance ?

Devriez-vous vous en soucier ?

Réponse : Que diriez-vous de System.out.println() ?

Le modèle de façade et le principe de moindre connaissance





Des outils pour votre boîte à outils de conception

Votre boîte à outils commence à être lourde ! Dans ce chapitre, nous avons ajouté quelques modèles qui nous permettent de modifier les interfaces et de réduire le couplage entre les clients et les systèmes qu'ils utilisent.

Notions de base sur OO

- Privilégiez la composition plutôt que l'héritage.
- Programmez vers des interfaces, pas vers des implémentations.
- Efforcez-vous d'obtenir des conceptions faiblement couplées entre les objets qui interagissent.
- Les cours devraient être ouverts pour prolongation mais fermés pour modification.
- Dépendez des abstractions. Ne dépendez pas de classes concrètes.
- Parlez uniquement à vos amis.

Principes OO

- Encapsuler ce qui varie.
- Abstraction
- Encapsulation
- Héritage

Modèles OO

- Stratégie - définit une famille d'algorithimes, encapsule et rend les rendes.
- Observateur - définit un accès à - beaucoup d'objets.
- Adaptateur - transforme l'interface d'un variante indépendamment de l'autre.
- Écoulement - échange de données longues de temps.
- Facade - fournit une interface unifiée à un ensemble d'interfaces dans un sous-système.

... et DEUX nouveaux modèles. Chacun modifie une interface, l'adaptateur pour convertir et la façade pour unifier et simplifier.

... et DEUX nouveaux modèles. Chacun modifie une interface, l'adaptateur pour convertir et la façade pour unifier et simplifier.



BULLET POINTS

f Lorsque vous devez utiliser une classe existante et que son interface n'est pas celle dont vous avez besoin, utilisez un adaptateur.

f Lorsque vous avez besoin de simplifier et d'unifier une grande interface ou un ensemble complexe d'interfaces, utilisez une façade.

f Un adaptateur transforme une interface en une interface attendue par un client.

f Une façade dissocie un client d'un sous-système complexe.

f L'implémentation d'un adaptateur peut nécessiter peu ou beaucoup de travail selon la taille et la complexité de l'interface cible.

f Implémentation d'une façade nécessite que nous componsons la façade avec son sous-système et que nous utilisions la délégation pour exécuter les travaux de façade.

f Il existe deux formes du modèle d'adaptateur : les adaptateurs d'objet et les adaptateurs de classe. Les adaptateurs de classe nécessitent un héritage multiple.

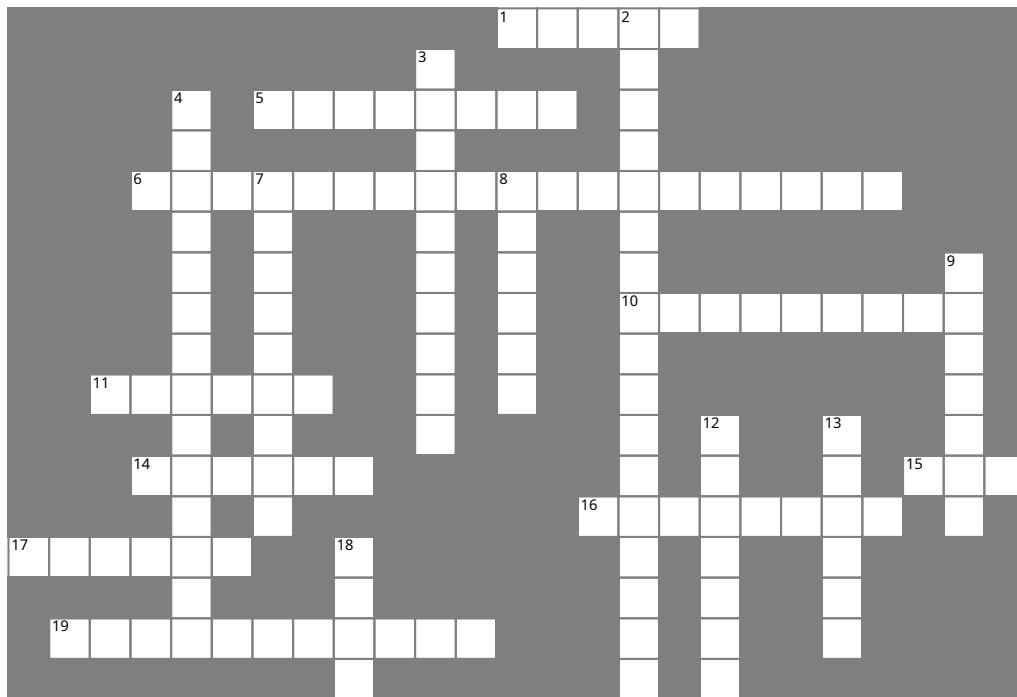
f Vous pouvez implémenter plusieurs façades pour un sous-système.

f Un adaptateur enveloppe un objet pour modifier son interface, un décorateur enveloppe un objet pour ajouter de nouveaux comportements et responsabilités, et une façade « enveloppe » un ensemble d'objets pour simplifier.



Mots croisés sur les modèles de conception

Oui, c'est un autre jeu de mots croisés. Tous les mots de la solution sont tirés de ce chapitre.



À TRAVERS

1. Vrai ou faux ? Les adaptateurs ne peuvent encapsuler qu'un seul objet.
5. Un adaptateur _____ une interface.
6. Film que nous avons regardé (cinq mots).
10. Si vous êtes en Grande-Bretagne, vous pourriez avoir besoin de l'un de ces (deux mots).
11. Adaptateur à deux rôles (deux mots).
14. La façade _____ toujours accès bas.
15. Les canards le font mieux que les dindes.
16. Inconvénient du principe de moindre connaissance : trop de _____.
17. Un _____ simplifie une interface.
19. Nouveau rêve américain (deux mots).

VERS LE BAS

2. Le décorateur a appelé cet adaptateur (trois mots).
3. Un avantage de la façade.
4. Un principe qui n'était pas aussi simple qu'il le paraissait (deux mots).
7. Un _____ ajoute un nouveau comportement.
8. Se faire passer pour un canard.
9. Exemple qui viole le principe de la moindre connaissance : System.out._____.
12. Aucun film n'est complet sans cela.
13. Le client adaptateur utilise l'interface _____.
18. On peut dire qu'un adaptateur et un décorateur _____ un objet.



Sharpen your pencil Solution

la classe publique DuckAdapter implémente la Turquie {

```
    Canard canard;
    Rand aléatoire;
    public DuckAdapter(Canard canard) {
        ce.canard = canard;
        rand = nouveau Aléatoire();
    }
    public void engloutir() {
        canard.coin-coin();
    }
    public void voler() {
        si (rand.nextInt(5) == 0) {
            canard.mouche();
        }
    }
}
```

Disons que nous avons également besoin d'un adaptateur qui convertit un canard en dinde. Appelons-le DuckAdapter. Voici notre solution :

Nous adaptons maintenant les dindes aux canards, nous implémentons donc l'interface Turquie.

Nous cachons une référence au Canard que nous adaptons.

Nous créons également un objet aléatoire ; jetez un œil à la méthode fly() pour voir comment elle est utilisée.

Un glouglou devient juste un charlatan.

Comme les canards volent beaucoup plus longtemps que les dindes, nous avons décidé de ne faire voler le canard qu'une fois sur cinq en moyenne.



Sharpen your pencil Solution

L'une ou l'autre de ces classes viole-t-elle le principe de la moindre connaissance ? Pourquoi ou pourquoi pas ?

```
Maison publique {
    Station Météo;
    // autres méthodes et constructeur
    public float getTemp() {
        station de retour.getThermometer().getTemperature();
    }
}
```

Viole le principe de la moindre connaissance ! Vous appelez la méthode d'un objet renvoyé par un autre appel.

```
Maison publique {
    Station Météo;
    // autres méthodes et constructeur
    public float getTemp() {
        Thermomètre thermomètre = station.getThermometer(); return
        getTempHelper(thermomètre);
    }

    public float getTempHelper(Thermomètre thermomètre) {
        retourner thermomètre.getTemperature();
    }
}
```

Cela ne viole pas le principe de la moindre connaissance ! Cela ressemble à une façon de contourner le principe. Est-ce que quelque chose a vraiment changé depuis que nous je viens de déplacer l'appel vers une autre méthode ?



Exercice Solution

Vous avez vu comment implémenter un adaptateur qui adapte une énumération à un itérateur ; écrivez maintenant un adaptateur qui adapte un itérateur à une énumération.

```
classe publique IteratorEnumeration implémente Enumeration<Object> {
    Itérateur<?> itérateur;
    public IteratorEnumeration(Iterator<?> itérateur) {
        this.iterator = itérateur;
    }

    public booléen hasMoreElements() {
        renvoyer itérateur.hasNext();
    }

    Objet public nextElement() {
        retourner itérateur.next();
    }
}
```

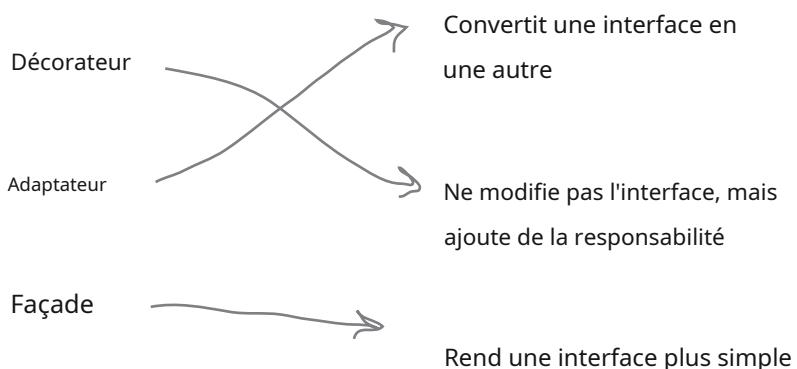
Notez que nous gardons le paramètre de type générique donc cela fonctionnera pour tout type d'objet.



Associez chaque modèle à son intention :

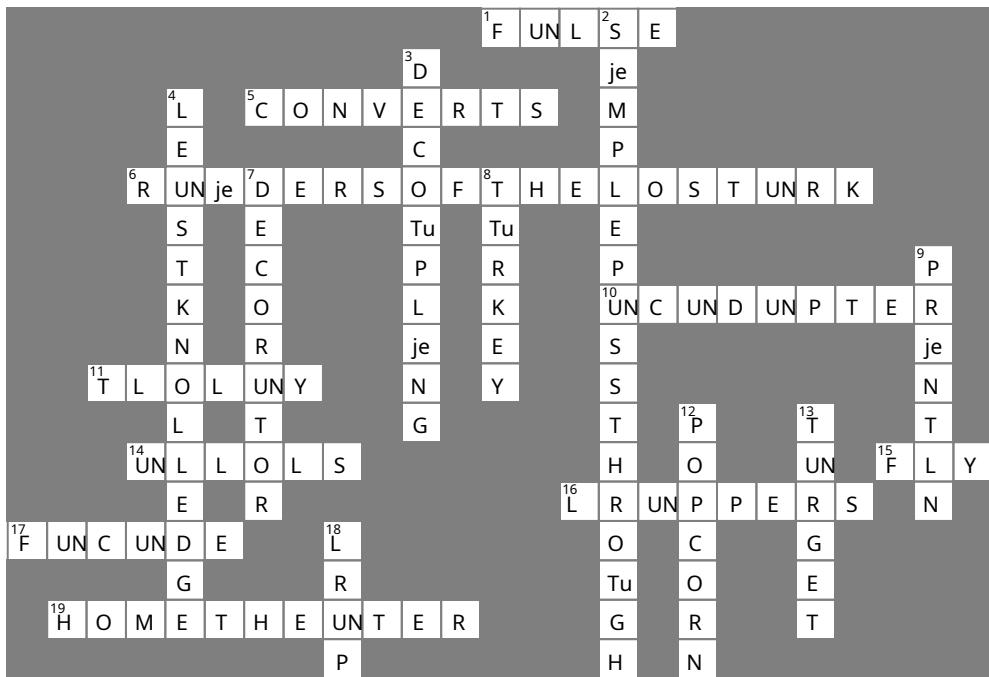
Modèle

Intention





Solution de mots croisés sur les modèles de conception



8 Le modèle de méthode de modèle

* Encapsulant * Algorithmes *

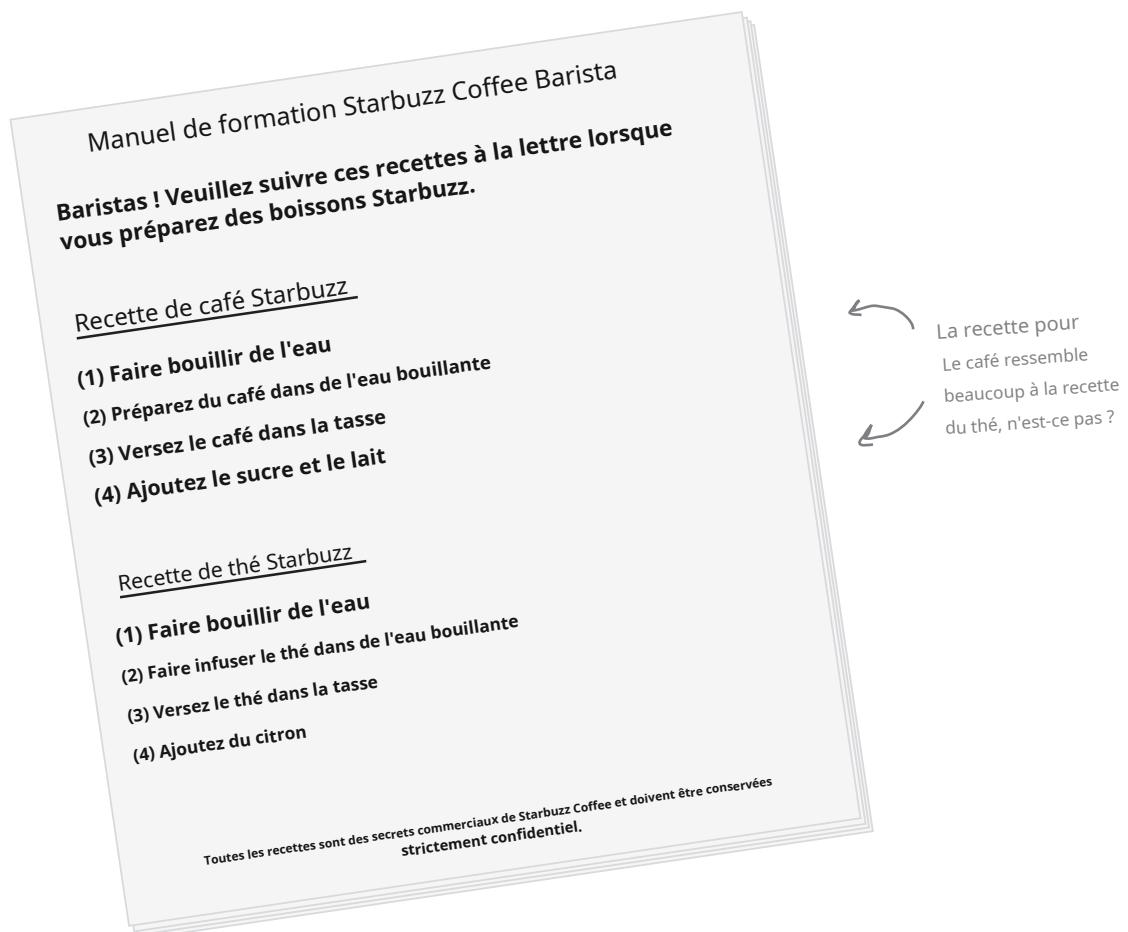


Nous sommes sur une lancée d'encapsulation ; nous avons encapsulé la création d'objets, l'invocation de méthodes, les interfaces complexes, les canards, pizzas...quelle pourrait être la prochaine étape ? Nous allons nous atteler à l'encapsulation de morceaux d'algorithme afin que les sous-classes puissent se connecter directement à un calcul à tout moment. Nous allons même découvrir un principe de conception inspiré d'Hollywood. Commençons...

Il est temps de prendre un peu plus de caféine

Certaines personnes ne peuvent pas vivre sans café,
d'autres ne peuvent pas vivre sans thé. L'ingrédient
commun ? La caféine, bien sûr !

Mais ce n'est pas tout : le thé et le café sont préparés de manière très
similaire. Voyons cela de plus près :



Préparation de quelques cours de café et de thé (en Java)

Jouons au « barista codeur » et écrivons du code pour créer du café et du thé.



Voici le café :

```

    Voici notre cours de café pour préparer du café.
    ↓
classe publique Café {

    void prepareRecipe() {
        faire bouillir l'eau();
        brewCoffeeGrinds();
        verserDansLaTasse();
        ajouterSucreEtLait();
    }

    public void faire bouillir l'eau() {
        System.out.println("Eau bouillante");
    }

    public void brewCoffeeGrinds() {
        System.out.println("Café qui coule à travers le filtre");
    }

    public void pourInCup() {
        System.out.println("Verser dans la tasse");
    }

    public void addSugarAndMilk() {
        System.out.println("Ajout de sucre et de lait");
    }
}

    Voici notre recette de café, directement
    tirée du manuel de formation.
    Chacune des étapes est implémentée comme
    une méthode distincte.

    Chacune de ces méthodes
    implémente une étape de
    l'algorithme. Il y a
    une méthode pour faire bouillir
    de l'eau, préparer le café, verser
    le café dans une tasse et ajouter
    du sucre et du lait.

```

Et maintenant le thé...



classe publique Thé {

```
void prepareRecipe() {  
    faire bouillir l'eau();  
    sachet de thé steep();  
    verserDansLaTasse();  
    ajouterCitron();  
}
```

Cela ressemble beaucoup à celui que nous venons d'implémenter dans Coffee ; les deuxième et quatrième étapes sont différentes, mais c'est fondamentalement la même recette.

```
public void faire bouillir l'eau() {  
    System.out.println("Eau bouillante");  
}
```



```
public void steepTeaBag() {  
    System.out.println("Infuser le thé");  
}
```

Ces deux-là
les méthodes sont
spécialisé dans le thé.

```
public void addLemon() {  
    System.out.println("Ajout de citron");  
}
```



```
public void pourInCup() {  
    System.out.println("Verser dans la tasse");  
}
```



}

Notez que ceux-ci
deux méthodes
sont exactement les
pareil qu'eux
sont dans le café !
Donc nous avons définitivement
avoir du code
duplication en cours
ici.

Lorsque nous avons du code en double,
c'est un bon signe que nous devons nettoyer la
conception. Il semble qu'ici nous devrions abstraire
les points communs dans une base
Le café et le thé sont très similaires.



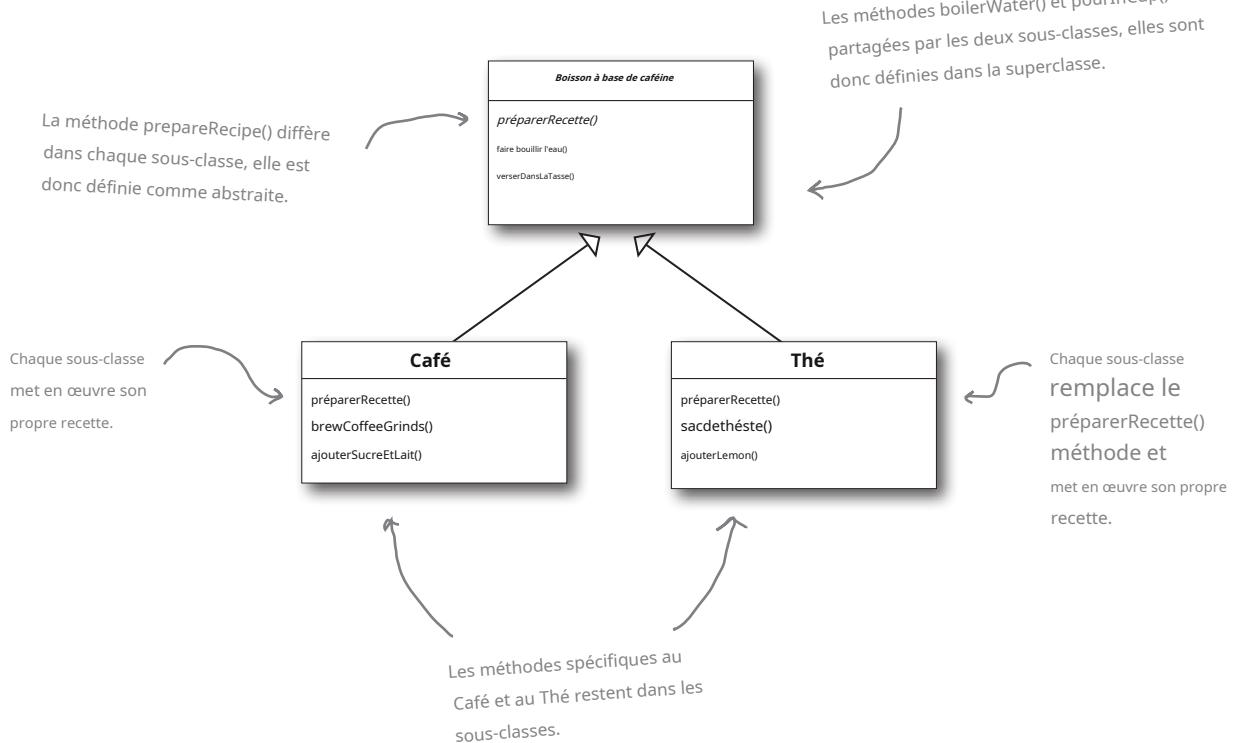


Puzzle de conception

Vous avez vu que les classes Coffee et Tea comportent une bonne partie de duplication de code. Jetez un autre coup d'œil aux classes Coffee et Tea et dessinez un diagramme de classes montrant comment vous repenseriez les classes pour supprimer la redondance :

Faisons abstraction du café et du thé

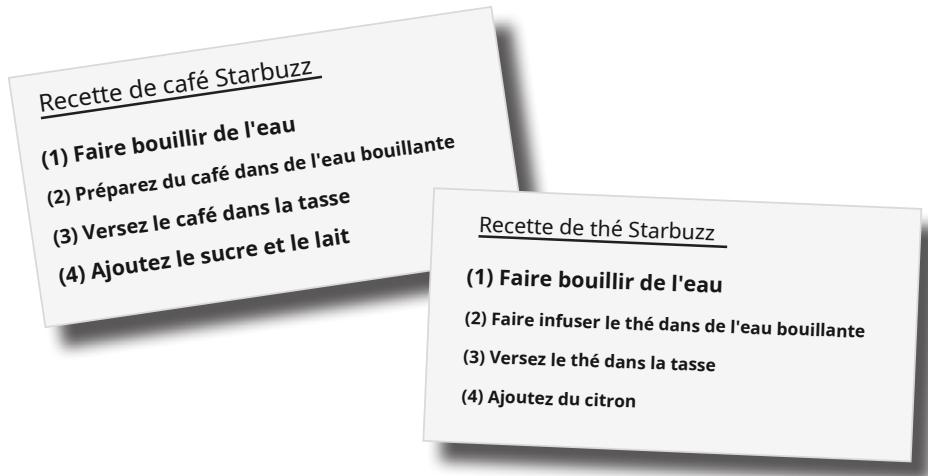
Il semble que nous ayons un exercice de conception assez simple à réaliser avec les classes Coffee et Tea. Votre première coupe aurait pu ressembler à ceci :



Avons-nous fait du bon travail sur la refonte ? Hmmmm, regardez à nouveau. Avons-nous négligé un autre point commun ? Quelles sont les autres similitudes entre le café et le thé ?

Aller plus loin dans le design...

Alors, qu'ont d'autre en commun le café et le thé ? Commençons par les recettes.



Notez que les deux recettes suivent le même algorithme :

- 1 faites bouillir de l'eau.
- 2 Utilisez l'eau chaude pour extraire le café ou le thé.
- 3 Versez la boisson obtenue dans une tasse.
- 4 Ajoutez les condiments appropriés à la boisson.

Ces deux-là sont déjà abstrait dans la classe de base.

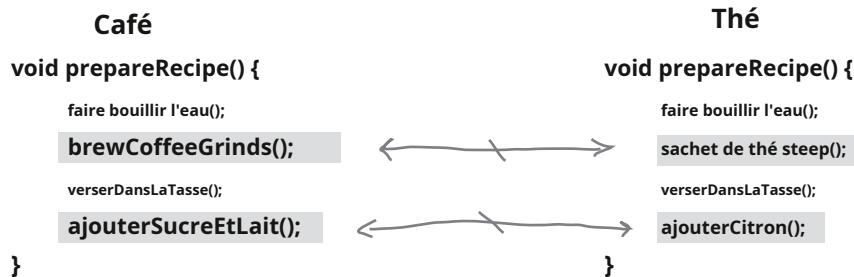
Ce ne sont pas abstrait mais sont les mêmes; ils s'appliquent simplement à différent boissons.

Alors, pouvons-nous trouver un moyen d'abstraire `prepareRecipe()` également ? Oui, découvrons-le...

Abstraction de prepareRecipe()

Passons en revue l'abstraction de `prepareRecipe()` de chaque sous-classe (c'est-à-dire les classes `Coffee` et `Tea`)...

- 1 Le premier problème que nous avons est que `Coffee` utilise les méthodes `brewCoffeeGrinds()` et `addSugarAndMilk()`, tandis que `Tea` utilise les méthodes `steepTeaBag()` et `addLemon()`.



Réfléchissons à cela : le trempage et l'infusion ne sont pas si différents ; ils sont plutôt analogues. Créons donc un nouveau nom de méthode, disons `brew()`, et nous utiliserons le même nom que nous préparions du café ou du thé.

De même, ajouter du sucre et du lait revient à peu près au même que d'ajouter un citron : les deux ajoutent des condiments à la boisson. Inventons également un nouveau nom de méthode, `addCondiments()`, pour gérer cela. Ainsi, notre nouvelle méthode `prepareRecipe()` ressemblera à ceci :

```
void prepareRecipe() {  
    faire bouillir l'eau();  
    brasser();  
    verserDansLaTasse();  
    ajouterCondiments();  
}
```

- 2 Nous avons maintenant une nouvelle méthode `prepareRecipe()`, mais nous devons l'intégrer au code. Pour ce faire, nous commencerons par la superclasse `CaffeineBeverage` :

(Code sur le
(page suivante.)

CaffeineBeverage est abstrait, tout comme dans la conception de la classe.

```
classe abstraite publique CaffeineBeverage {
```

```
final void prepareRecipe() {
    faire bouillir l'eau();
    brasser();
    verserDansLaTasse();
    ajouterCondiments();
}
```

Maintenant, la même méthode prepareRecipe() sera utilisée pour préparer à la fois du thé et du café. prepareRecipe() est déclarée final car nous ne voulons pas que nos sous-classes puissent remplacer cette méthode et modifier la recette ! Nous avons généralisé les étapes 2 et 4 à brew() la boisson et addCondiments().

abstrait void brew();

abstrait void addCondiments();

Comme Coffee et Tea gèrent ces méthodes de manière différente, elles vont devoir être déclarées comme abstraites. Laissez les sous-classes s'en occuper !

```
void faire bouillir l'eau() {
    System.out.println("Eau bouillante");
}
```

N'oubliez pas que nous les avons déplacés dans la classe CaffeineBeverage (dans notre diagramme de classes).

```
void pourInCup() {
    System.out.println("Verser dans la tasse");
}
```

}

- 3 Enfin, nous devons nous occuper des classes de café et de thé. Ils comptent désormais sur CaffeineBeverage pour gérer la recette, il leur suffit donc de gérer l'infusion et les condiments :

```
classe publique Thé prolonge la boisson à la caféine {
    public void brew() {
        System.out.println("Infuser le thé");
    }
    public void addCondiments() {
        System.out.println("Ajout de citron");
    }
}
```

Comme dans notre conception, le thé et le café étendent désormais CaffeineBeverage.

Le thé doit définir brew() et addCondiments(), les deux méthodes abstraites de CaffeineBeverage.

C'est pareil pour le café, sauf que le café contient du café, du sucre et du lait au lieu de sachets de thé et de citron.

```
classe publique Café prolonge la boisson à la caféine {
    public void brew() {
        System.out.println("Café qui coule à travers le filtre");
    }
    public void addCondiments() {
        System.out.println("Ajout de sucre et de lait");
    }
}
```



Sharpen your pencil

Dessinez le nouveau diagramme de classe maintenant que nous avons déplacé l'implémentation de `prepareRecipe()` dans la classe `CaffeineBeverage` :

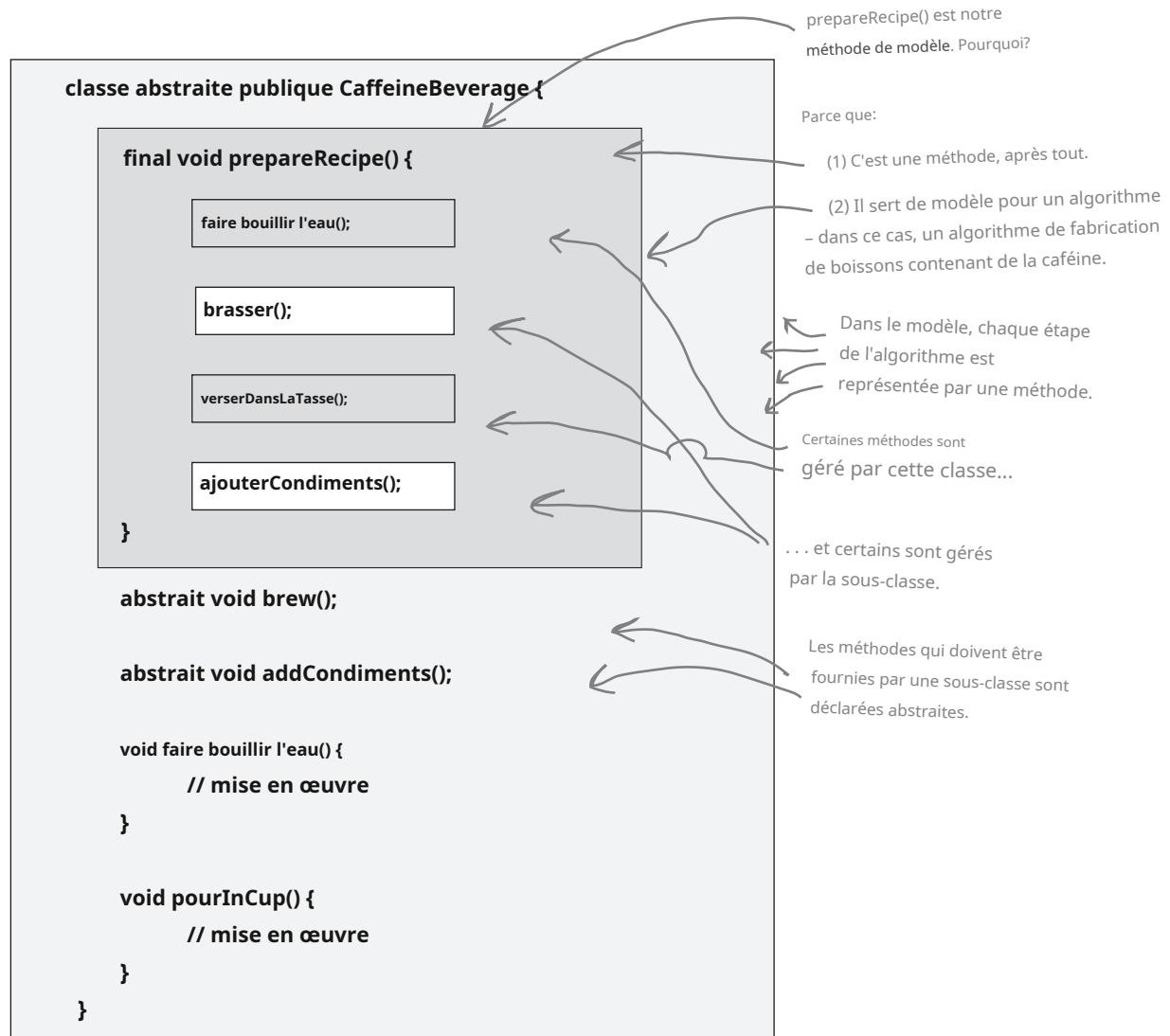
Qu'avons-nous fait ?



Découvrez la méthode du modèle

Nous venons tout juste d'implémenter le modèle de méthode de modèle. Qu'est-ce que c'est ?

Regardons la structure de la classe CaffeineBeverage ; elle contient la « méthode de modèle » réelle :



La méthode modèle définit les étapes d'un algorithme et permet aux sous-classes de fournir l'implémentation d'une ou plusieurs étapes.

Préparons du thé...

Passons maintenant à la préparation du thé et décrivons le fonctionnement de la méthode du modèle. Vous verrez que la méthode du modèle contrôle l'algorithme ; à certains moments de l'algorithme, elle laisse la sous-classe fournir l'implémentation des étapes...

Derrière
les scènes



1

Ok, nous avons d'abord besoin d'un objet Thé...

```
Thé monThé = nouveau Thé();
```

```
faire bouillir l'eau();
brasser();
verserDansLaTasse();
ajouterCondiments();
```

2

Ensuite, nous appelons la méthode template :

```
monThé.préparerRecette();
```

qui suit l'algorithme de fabrication des boissons à la caféine...

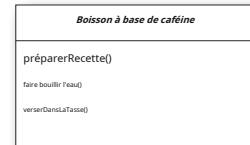
La méthode `préparerRecette()` contrôle l'algorithme. Personne ne peut changer cela, et il compte sur les sous-classes pour fournir une partie ou la totalité de l'implémentation.

3

Tout d'abord, nous faisons bouillir de l'eau :

```
faire bouillir l'eau();
```

ce qui se passe dans CaffeineBeverage.



4

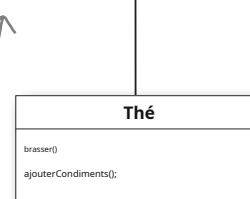
Ensuite, nous devons préparer le thé, ce que seule la sous-classe sait faire :

```
brasser();
```

5

Maintenant, nous versons le thé dans la tasse ; c'est la même chose pour toutes les boissons, donc cela se passe dans CaffeineBeverage :

```
verserDansLaTasse();
```



6

Enfin, nous ajoutons les condiments, qui sont spécifiques à chaque boisson, donc la sous-classe implémente ceci :

```
ajouterCondiments();
```

Qu'est-ce que la méthode Template nous a apporté ?



Thé et café sous-alimentés
mise en œuvre



Nouvelle boisson à la caféine branchée
alimentée par Template Method

Le café et le thé mènent la danse ; ils contrôlent l'algorithme.

La classe CaffeineBeverage dirige le spectacle ; elle possède l'algorithme et le protège.

Le code est dupliqué dans Coffee et Tea.

La classe CaffeineBeverage maximise la réutilisation parmi les sous-classes.

Les modifications du code de l'algorithme nécessitent l'ouverture des sous-classes et la réalisation de plusieurs modifications.

L'algorithme réside à un seul endroit et les modifications de code ne doivent être effectuées qu'à cet endroit.

Les cours sont organisés selon une structure qui nécessite beaucoup de travail pour ajouter une nouvelle boisson contenant de la caféine.

Le modèle de méthode de modèle fournit un cadre que d'autres cafétiers Les boissons peuvent être branchées. Les nouvelles boissons à la caféine n'ont besoin que de quelques méthodes.

La connaissance de l'algorithme et de la manière de le mettre en œuvre est répartie sur de nombreuses classes.

La classe CaffeineBeverage concentre les connaissances sur l'algorithme et s'appuie sur des sous-classes pour fournir des implémentations complètes.

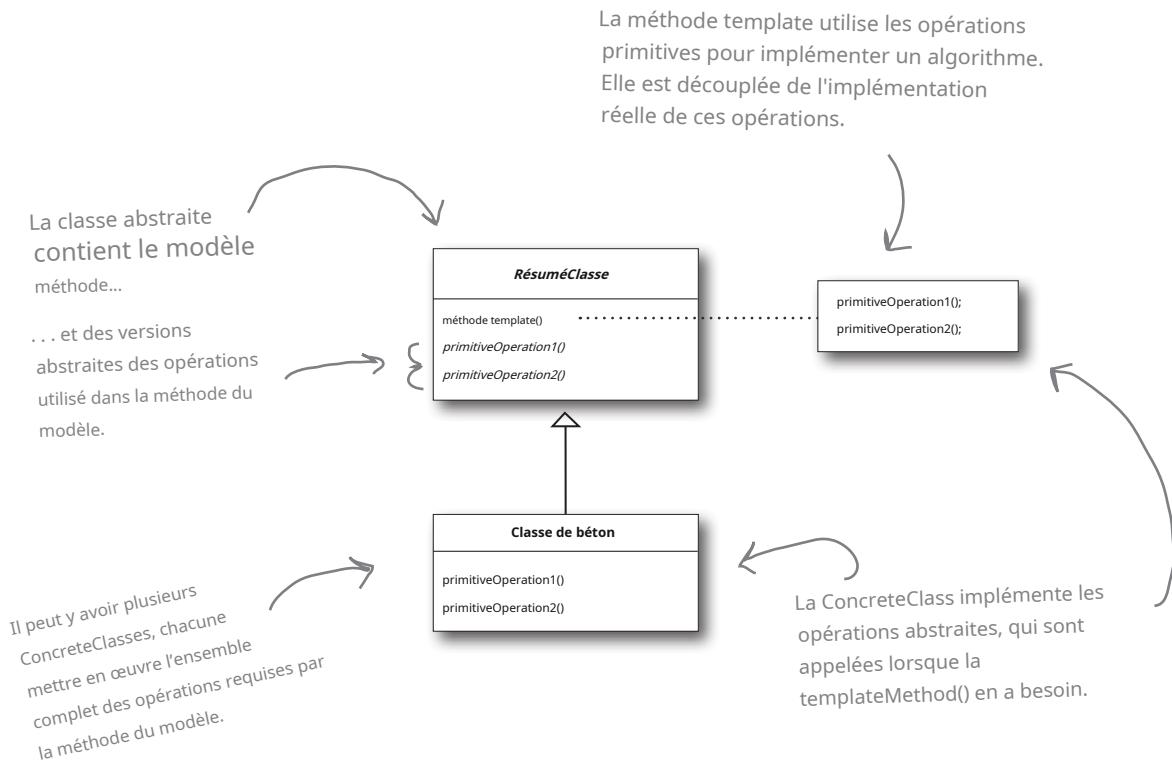
Modèle de méthode Modèle défini

Vous avez vu comment fonctionne le modèle de méthode de modèle dans notre exemple de thé et de café ; consultez maintenant la définition officielle et notez tous les détails :

Le modèle de méthode de modèle définit le squelette d'un algorithme dans une méthode, en déléguant certaines étapes à des sous-classes. La méthode Template permet aux sous-classes de redéfinir certaines étapes d'un algorithme sans modifier la structure de l'algorithme.

Ce modèle consiste à créer un modèle pour un algorithme. Qu'est-ce qu'un modèle ? Comme vous l'avez vu, il s'agit simplement d'une méthode ; plus précisément, il s'agit d'une méthode qui définit un algorithme comme un ensemble d'étapes. Une ou plusieurs de ces étapes sont définies comme abstraites et implémentées par une sous-classe. Cela garantit que la structure de l'algorithme reste inchangée, tandis que les sous-classes fournissent une partie de l'implémentation.

Regardons le diagramme de classe :





Le code de près

Examinons de plus près comment la classe `AbstractClass` est définie, y compris la méthode de modèle et les opérations primitives.

Nous avons ici notre classe abstraite ; elle est déclarée abstraite et destinée à être sous-classée par des classes qui fournissent des implémentations des opérations.

```
classe abstraite AbstractClass {
```

```
    modèle final void() {
        primitiveOperation1();
        primitiveOperation2();
        opérationconcrète();
    }
```

```
    abstrait void primitiveOperation1();
```

```
    abstrait void primitiveOperation2();
```

```
    void opérationconcrète() {
        // implémentation ici
    }
}
```

Voici la méthode du modèle. Elle est déclarée finale pour empêcher les sous-classes de retravailler la séquence d'étapes de l'algorithme.

La méthode du modèle définit la séquence d'étapes, chacune représentée par une méthode.

Dans cet exemple, deux des opérations primitives doivent être implémentées par des sous-classes concrètes.

Nous avons également une opération concrète définie dans la classe abstraite. Celle-ci pourrait être remplacée par des sous-classes, ou nous pourrions empêcher la substitution en déclarant `concreteOperation()` comme final. Plus d'informations à ce sujet dans un instant...



Le code de près

Nous allons maintenant examiner de plus près les types de méthodes qui peuvent figurer dans la classe abstraite :

Nous avons changé le templateMethod() à inclure un nouvel appel de méthode.

```
classe abstraite AbstractClass {

    modèle final void() {
        primitiveOperation1();
        primitiveOperation2();
        opérationconcrète();
        crochét();
    }

    abstrait void primitiveOperation1();

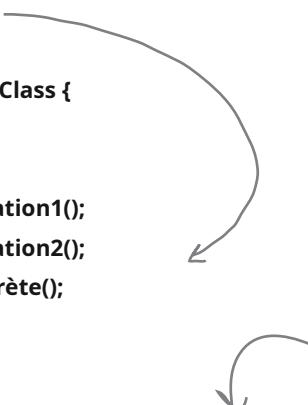
    abstrait void primitiveOperation2();

    final void opérationconcrète() {
        // implémentation ici
    }

    void crochét() {}

}
```

Une méthode concrète, mais qui ne sert à rien !



Nous avons toujours nos méthodes de fonctionnement primitives ; ceux-ci sont abstraits et implémentés par des sous-classes concrètes.

Une opération concrète est définie dans la classe abstraite. Celle-ci est déclarée finale afin que les sous-classes ne puissent pas la surcharger. Elle peut être utilisée directement dans la méthode template, ou utilisée par les sous-classes.



Nous pouvons également avoir des méthodes concrètes qui ne font rien par défaut ; nous les appelons des « hooks ». Les sous-classes sont libres de les remplacer, mais ne sont pas obligées de le faire. Nous allons voir à quel point elles sont utiles sur la page suivante.



Accro à Méthode de modèle...

Un hook est une méthode déclarée dans la classe abstraite, mais à laquelle on ne donne qu'une implémentation vide ou par défaut. Cela donne aux sous-classes la possibilité de « se connecter » à l'algorithme à différents moments, si elles le souhaitent ; une sous-classe est également libre d'ignorer le hook.

Il existe plusieurs utilisations des hooks ; examinons-en une maintenant. Nous parlerons de quelques autres utilisations plus tard :

```
classe abstraite publique CaffeineBeverageWithHook {
```

```
    final void prepareRecipe() {
        faire bouillir l'eau();
        brasser();
        verserDansLaTasse();
        si (clientVeutCondiments()) {
            ajouterCondiments();
        }
    }
```

```
    abstrait void brew();
```

```
    abstrait void addCondiments();
```

```
    void faire bouillir l'eau() {
        System.out.println("Eau bouillante");
    }
```

```
    void pourInCup() {
        System.out.println("Verser dans la tasse");
    }
```

```
    booléen customerWantsCondiments() {
        renvoie vrai ;
    }
```

Avec un hook, je peux surcharger la méthode ou non. C'est mon choix. Si je ne le fais pas, la classe abstraite fournit une implémentation par défaut.



Nous avons ajouté une petite instruction conditionnelle qui base son succès sur une méthode concrète, `customerWantsCondiments()`. Si le client VEUT des condiments, alors seulement nous appelons `addCondiments()`.

Nous avons défini ici une méthode avec une implémentation par défaut (essentiellement) vide. Cette méthode renvoie simplement true et ne fait rien d'autre.

C'est un *crochet* parce que la sous-classe peut remplacer cette méthode, mais n'est pas obligée de le faire.

Utilisation du crochet

Pour utiliser le hook, nous le remplaçons dans notre sous-classe. Ici, le hook contrôle si la classe CaffeineBeverage évalue une certaine partie de l'algorithme, c'est-à-dire si elle ajoute un condiment à la boisson.

Comment savoir si le client veut le condiment ? Il suffit de lui demander !

```
classe publique CoffeeWithHook étend CaffeineBeverageWithHook {
```

```
    public void brew() {
        System.out.println("Café qui coule à travers le filtre");
    }
```

```
    public void addCondiments() {
        System.out.println("Ajout de sucre et de lait");
    }
```

```
    public boolean customerWantsCondiments() {
```

```
        Chaîne de réponse = getUserInput();

        si (réponse.toLowerCase().startsWith("y")) {
            renvoie vrai ;
        } autre {
            retourner faux;
        }
    }
```

```
    chaîne privée getUserInput() {
        Chaîne de réponse = null;
```

```
        System.out.print("Voulez-vous du lait et du sucre avec votre café (y/n) ? ");
```

```
        BufferedReader dans = nouveau BufferedReader(nouveau InputStreamReader(System.in));
        essayez {
```

```
            réponse = in.readLine(); }
```

```
        catch (IOException ioe) {
            System.err.println("Erreur d'E/S lors de la tentative de lecture de votre réponse");
        }
```

```
        si (réponse == null) {
            retourner "non";
        }
```

```
        retourner la réponse;
    }
```

C'est ici que vous remplacez le hook et fournissez votre propre fonctionnalité.

Obtenez l'avis de l'utilisateur sur le choix des condiments et renvoie vrai ou faux, selon l'entrée.

↑ Ce code demande si l'utilisateur souhaite du lait et du sucre et obtient l'entrée à partir de la ligne de commande.

```
}
```

Faisons un essai routier

Ok, l'eau bout... Voici le code de test où nous créons un thé chaud et un café chaud.

```
classe publique BeverageTestDrive {  
    public static void main(String[] args) {  
  
        ThéAvecHook teaHook = nouveau ThéAvecHook(); ← Crée un thé.  
        CaféAvecHook coffeeHook = nouveau CaféAvecHook(); ← Crée un café.  
  
        System.out.println("\nPréparer le thé...");  
        teaHook.prepareRecipe();  
  
        System.out.println("\nPréparer le café...");  
        coffeeHook.prepareRecipe();  
    }  
}
```

Et essayons...

```
Aide sur la fenêtre d'édition de fichier send-more-honesttea  
% java BeverageTestDrive  
  
Faire du thé...  
Eau bouillante  
Faire infuser le thé  
Verser dans la tasse  
Voulez-vous du citron avec votre thé (t/n) ? y Ajouter ← Une tasse de thé fumante, et oui,  
du citron bien sûr, nous voulons ce citron !  
  
Faire du café...  
Eau bouillante  
Café s'écoulant à travers le filtre  
Versé dans la tasse  
Voulez-vous du lait et du sucre avec votre café (y/n) ? n ← Et une bonne tasse de café chaud, mais  
% nous renoncerons aux condiments qui font grossir la taille.
```



Vous savez quoi ? Nous sommes d'accord avec vous. Mais vous devez admettre qu'avant d'y penser, c'était un exemple assez intéressant de la façon dont un hook peut être utilisé pour contrôler conditionnellement le flux de l'algorithme dans la classe abstraite. N'est-ce pas ?

Nous sommes sûrs que vous pouvez penser à de nombreux autres scénarios plus réalistes dans lesquels vous pourriez utiliser la méthode de modèle et les hooks dans votre propre code.

there are no Dumb Questions

Q: Lorsque je crée une méthode de modèle, comment savoir quand utiliser des méthodes abstraites et quand utiliser des hooks ?

UN: Utilisez des méthodes abstraites lorsque votre sous-classe DOIT fournir une implémentation de la méthode ou de l'étape de l'algorithme. Utilisez des hooks lorsque cette partie de l'algorithme est facultative. Avec les hooks, une sous-classe peut choisir d'implémenter ce hook, mais elle n'est pas obligée de le faire.

Q: À quoi servent réellement les crochets ?

UN: Les hooks ont plusieurs utilisations. Comme nous venons de le dire, un hook peut permettre à une sous-classe d'implémenter une partie facultative d'un algorithme, ou si cela n'est pas important pour l'implémentation de la sous-classe, elle peut l'ignorer. Une autre utilisation consiste à donner à la sous-classe une chance de réagir à une étape de la méthode de modèle qui est sur le point de se produire ou qui vient de se produire. Par exemple, une méthode hook comme justReorderedList() permet à la sous-classe d'effectuer une activité (comme le réaffichage d'une représentation à l'écran) après qu'une liste interne a été réorganisée. Comme vous l'avez vu, un hook peut également fournir à une sous-classe la possibilité de prendre une décision pour la classe abstraite.

Q: Une sous-classe doit-elle implémenter toutes les méthodes abstraites de la AbstractClass ?

UN: Oui, chaque sous-classe concrète définit l'ensemble des méthodes abstraites et fournit une implémentation complète des étapes non définies de l'algorithme de la méthode modèle.

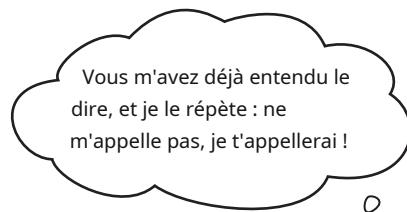
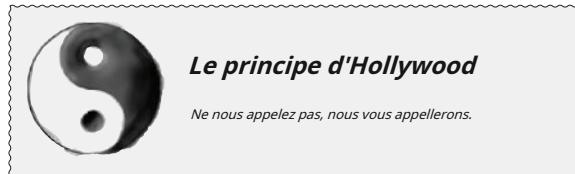
Q: Il semble que je devrais garder mes méthodes abstraites en petit nombre ; sinon, ce sera un gros travail de les implémenter dans la sous-classe.

UN: C'est une bonne chose à garder à l'esprit lorsque vous écrivez des méthodes de modèle. Parfois, vous pouvez y parvenir en ne rendant pas les étapes de votre algorithme trop granulaires. Mais c'est évidemment un compromis : moins il y a de granularité, moins il y a de flexibilité.

N'oubliez pas également que certaines étapes seront facultatives, vous pouvez donc les implémenter sous forme de hooks plutôt que de méthodes abstraites, allégeant ainsi la charge sur les sous-classes de votre classe abstraite.

Le principe d'Hollywood

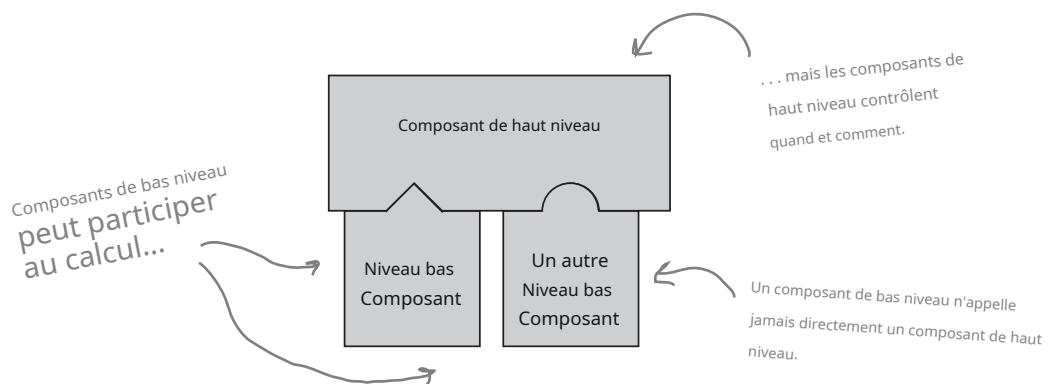
Nous avons un autre principe de conception pour vous ; il s'appelle le principe Hollywood :



Facile à retenir, n'est-ce pas ? Mais quel est le rapport avec la conception OO ?

Le principe d'Hollywood nous donne un moyen d'empêcher la « pourriture des dépendances ». La pourriture des dépendances se produit lorsque des composants de haut niveau dépendent de composants de bas niveau qui dépendent de composants de haut niveau qui dépendent de composants latéraux qui dépendent de composants de bas niveau, et ainsi de suite. Lorsque la pourriture s'installe, personne ne peut facilement comprendre la façon dont un système est conçu.

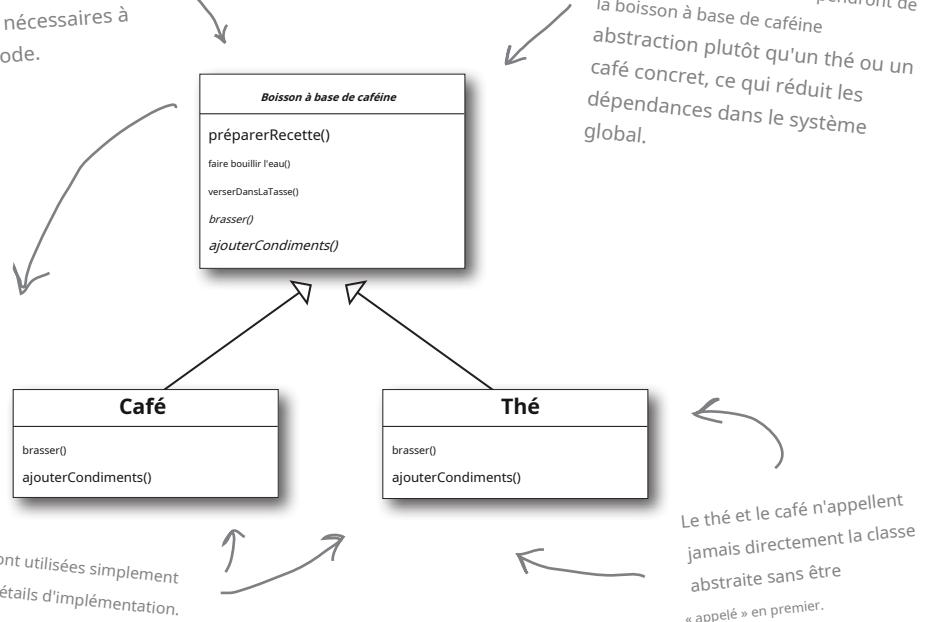
Avec le principe Hollywood, nous permettons aux composants de bas niveau de se connecter à un système, mais les composants de haut niveau déterminent quand ils sont nécessaires et comment. En d'autres termes, les composants de haut niveau donnent aux composants de bas niveau le traitement « ne nousappelez pas, nous vous appellerons ».



Le principe Hollywood et la méthode des modèles

Le lien entre le principe Hollywood et le modèle de méthode de modèle est probablement assez évident : lorsque nous concevons avec le modèle de méthode de modèle, nous disons aux sous-classes : « Ne nousappelez pas, nous vous appellerons. » Comment ? Jetons un autre coup d'œil à notre conception de CaffeineBeverage :

CaffeineBeverage est notre composant de haut niveau. Il contrôle l'algorithme de la recette et appelle les sous-classes uniquement lorsqu'elles sont nécessaires à l'implémentation d'une méthode.



Quels autres modèles utilisent le principe d'Hollywood ?

La méthode Factory et Observer ; et d'autres ?

there are no
Dumb Questions

Q:

Quel est le rapport entre le principe d'Hollywood et le principe d'inversion de dépendance que nous avons appris il y a quelques chapitres ?

UN:

Le principe d'inversion des dépendances nous apprend à éviter l'utilisation de classes concrètes et à travailler autant que possible avec des abstractions. Le principe d'Hollywood est une technique permettant de créer des frameworks ou des composants de manière à ce que les composants de niveau inférieur puissent être intégrés au calcul, mais sans créer de dépendances entre les composants de niveau inférieur et les couches de niveau supérieur. Ils ont donc tous deux pour objectif le découplage, mais le principe d'inversion des dépendances fait une déclaration beaucoup plus forte et plus générale sur la manière d'éviter les dépendances dans la conception.

Le principe d'Hollywood nous donne une technique pour créer des conceptions qui permettent aux structures de bas niveau d'interopérer tout en empêchant les autres classes de devenir trop dépendantes d'elles.

Q:

Un composant de bas niveau n'est-il pas autorisé à appeler une méthode dans un composant de niveau supérieur ?

UN:

Pas vraiment. En fait, un composant de bas niveau finira souvent par appeler une méthode définie au-dessus de lui dans la hiérarchie d'héritage uniquement par héritage. Mais nous voulons éviter de créer des dépendances circulaires explicites entre le composant de bas niveau et ceux de haut niveau.



Associez chaque motif à sa description :

Modèle

Méthode de modèle

Stratégie

Méthode d'usine

Description

Encapsulez les comportements interchangeables et utilisez la délégation pour décider quel comportement utiliser.

Les sous-classes décident comment pour implémenter des étapes dans un algorithme.

Les sous-classes décident quelles classes concrètes instancier.



Méthodes de modèles dans la nature

Le modèle de méthode de modèle est un modèle très courant et vous en trouverez beaucoup dans la nature. Vous devez cependant avoir l'œil vif, car il existe de nombreuses implémentations des méthodes de modèle qui ne ressemblent pas tout à fait à la conception classique du modèle.

Ce modèle apparaît si souvent parce qu'il s'agit d'un excellent outil de conception pour créer des frameworks, où le framework contrôle la manière dont quelque chose est fait, mais vous laisse (la personne utilisant le framework) spécifier vos propres détails sur ce qui se passe réellement à chaque étape de l'algorithme du framework.

Faisons un petit safari à travers quelques utilisations dans la nature (enfin, d'accord, dans l'API Java)...

En formation, nous étudions les modèles classiques. Cependant, lorsque nous sommes dans le monde réel, nous devons apprendre à reconnaître les modèles hors contexte. Nous devons également apprendre à reconnaître les variations de motifs, car dans le monde réel, un trou carré est pas toujours vraiment carré.





Tri avec la méthode du modèle

Quelle est la chose que nous devons souvent faire avec les tableaux ? Les trier !

Conscients de cela, les concepteurs de la classe Java Arrays nous ont fourni une méthode de modèle pratique pour le tri. Voyons comment fonctionne cette méthode :

Nous avons en fait deux méthodes ici et elles agissent ensemble pour fournir la fonctionnalité de tri.

Nous avons un peu simplifié ce code pour le rendre plus facile à expliquer. Si vous souhaitez tout voir, récupérez le code source Java et jetez-y un œil...

La première méthode, `sort()`, est simplement une méthode d'assistance qui crée une copie du tableau et la transmet comme tableau de destination à la méthode `mergeSort()`. Elle transmet également la longueur du tableau et indique au tri de commencer au premier élément.

```
public static void sort(Object[] a) {
    Objet aux[] = (Objet[])a.clone();
    mergeSort(aux, a, 0, a.length, 0);
}
```

La méthode `mergeSort()` contient l'algorithme de tri et s'appuie sur une implémentation de la méthode `compareTo()` pour compléter l'algorithme. Si vous êtes intéressé par le déroulement du tri, vous voudrez consulter le code source Java.

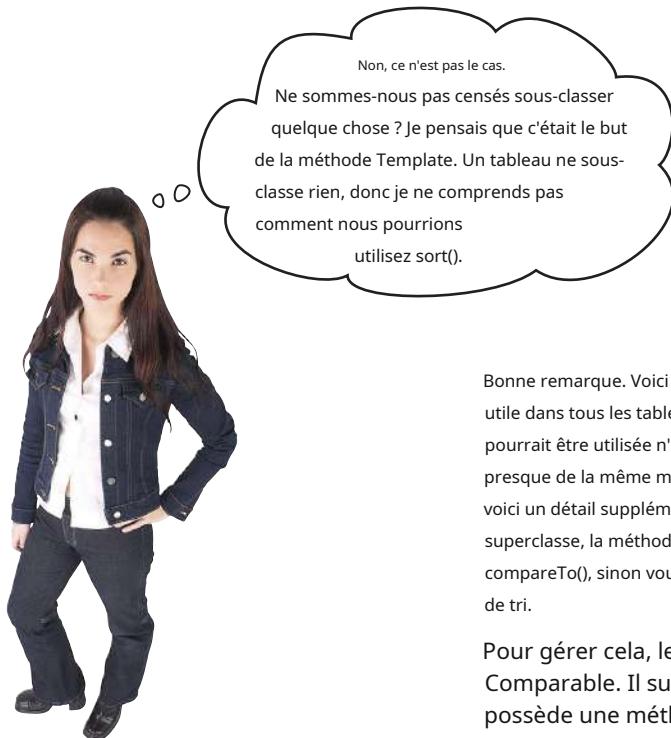
```
privé statique void mergeSort(Objet src[], Objet dest[],
                             int bas, int haut, int éteint)
{
    // beaucoup d'autres codes ici pour
    (int i=low; i<high; i++){
        pour (int j=i; j>faible &&
              ((Comparable)dest[j-1]).compareTo((Comparable)dest[j])>0; j--)
        {
            échanger(dest, j, j-1);
        }
    }
    // et beaucoup d'autres codes ici
}
```

Considérez ceci comme la méthode de modèle.

`compareTo()` est la méthode que nous devons implémenter pour « remplir » la méthode du modèle.

Nous avons des canards à trier...

Supposons que vous ayez un tableau de canards que vous souhaitez trier. Comment faites-vous ? Eh bien, la méthode de modèle `sort()` dans `Arrays` nous donne l'algorithme, mais vous devez lui dire comment comparer les canards, ce que vous faites en implémentant la méthode `compareTo()`... Cela a du sens ?

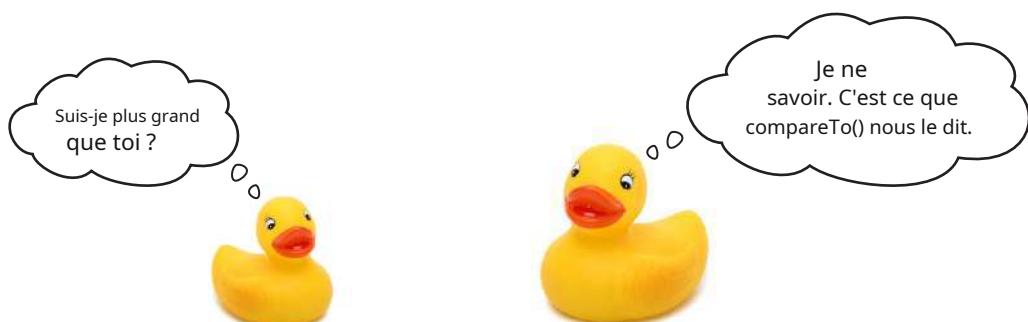


Bonne remarque. Voici le problème : les concepteurs de `sort()` voulaient qu'elle soit utile dans tous les tableaux, ils ont donc dû faire de `sort()` une méthode statique qui pourrait être utilisée n'importe où. Mais ce n'est pas grave, car elle fonctionne presque de la même manière que si elle était dans une superclasse. Maintenant, voici un détail supplémentaire : comme `sort()` n'est pas vraiment défini dans notre superclasse, la méthode `sort()` doit savoir que vous avez implémenté la méthode `compareTo()`, sinon vous n'avez pas la pièce nécessaire pour terminer l'algorithme de tri.

Pour gérer cela, les concepteurs ont utilisé l'interface `Comparable`. Il suffit d'implémenter cette interface, qui possède une méthode (surprise) : `compareTo()`.

Qu'est-ce que `compareTo()` ?

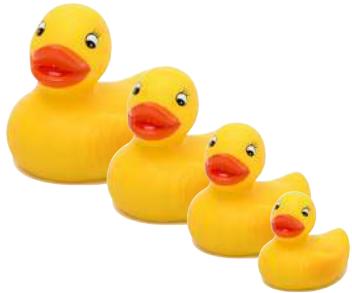
La méthode `compareTo()` compare deux objets et renvoie si l'un est inférieur, supérieur ou égal à l'autre. `sort()` utilise ceci comme base de sa comparaison des objets dans le tableau.



exécution comparable

Comparaison entre canards et canards

Ok, donc vous savez que si vous voulez trier les canards, vous allez devoir implémenter cette méthode `compareTo()` ; en faisant cela, vous donnerez à la classe `Arrays` ce dont elle a besoin pour compléter l'algorithme et trier vos canards.



Voici l'implémentation du canard :



N'oubliez pas que nous devons implémenter l'interface Comparable puisque nous ne sous-classons pas vraiment.

```
classe publique Duck implémente Comparable<Duck> {
```

```
    Nom de la chaîne ;
```

```
    int poids;
```



Nos canards ont un nom et un poids.

```
public Duck(Chaîne nom, int poids) {  
    ceci.nom = nom;  
    ceci.poids = poids;  
}
```

```
chaîne publique toString() {  
    retourner nom + " pèse " + poids;  
}
```



Nous gardons les choses simples ; tout ce que font les canards est d'imprimer leur nom et leur poids !

```
public int compareTo(Canard autreCanard) {  
  
    si (ceci.poids < autreCanard.poids) {  
        retourner -1;  
    } sinon si (ceci.poids == autreCanard.poids) {  
        retourner 0;  
    } else { // ceci.poids > autreCanard.poids  
        retour 1;  
    }  
}
```



compareTo() prend un autre canard pour comparer CE canard.



C'est ici que nous spécifions comment les canards se comparent. Si CE canard pèse moins que l'autre canard, nous renvoyons -1 ; s'ils sont égaux, nous renvoyons 0 ; et si CE canard pèse plus, nous renvoyons 1.

Trions quelques canards

Voici l'essai routier pour trier les canards...

```

classe publique DuckSortTestDrive {

    public static void main(String[] args) {
        Canard[] canards = {
            nouveau canard ("Daffy", 8),
            nouveau Canard ("Dewey", 2),
            nouveau canard ("Howard", 7),
            nouveau canard ("Louie", 2),
            nouveau Canard("Donald", 10),
            nouveau Canard("Huey", 2)
        };
    System.out.println("Avant le tri :");
    display(canards);                                ← Imprimons-les pour voir
                                                       leurs noms et leurs poids.

    Tableaux.sort(canards);                         ← C'est l'heure du tri !
                                                       Imprimons-les (à nouveau) pour voir
                                                       leurs noms et leurs poids.

    System.out.println("\nAprès le tri :");
    display(canards);
}
public static void display(Duck[] canards) {
    pour (Canard d : canards) {
        Système.out.println(d);
    }
}

```

Que le tri commence !

```

Fenêtre d'édition de fichier Aide DonaldNeedsToGoOnADiet
%java DuckSortTestDrive
Avant le tri :
Daffy pèse 8
Dewey pèse 2
Howard pèse 7
Louie pèse 2
Donald pèse 10
Huey pèse 2

Les canards non triés

Après le tri :
Dewey pèse 2
Louie pèse 2
Huey pèse 2
Howard pèse 7
Daffy pèse 8
Donald pèse 10
%

Les canards triés
%
```

La fabrication de la machine à trier les canards

Traçons à travers comment fonctionne la méthode de modèle `Arrays sort()`.

Nous allons voir comment la méthode de modèle contrôle l'algorithme et, à certains moments de l'algorithme, comment elle demande à nos Canards de fournir l'implémentation d'une étape...

**Derrière
les scènes**



1

Tout d'abord, nous avons besoin d'un tableau de canards :

`Canard[] canards = {nouveau Canard("Daffy", 8), ...};`

2

Ensuite, nous appelons la méthode de modèle `sort()` dans la classe `Arrays` et lui passons nos canards :

`Tableaux.sort(canards);`

La méthode `sort()` (et son assistant, `mergeSort()`) contrôlent la procédure de tri.

3

Pour trier un tableau, vous devez comparer deux éléments un par un jusqu'à ce que la liste entière soit dans l'ordre trié.

Lorsqu'il s'agit de comparer deux canards, la méthode `sort()` s'appuie sur la méthode `compareTo()` du canard pour savoir comment procéder. La méthode `compareTo()` est appelée sur le premier canard et transmise au canard à comparer :

`canards[0].compareTo(canards[1]);`

Premier canard

Canard pour le comparer à

Canard

`comparerÀ()`
`toString()`

→

Pas d'héritage,
contrairement à un typique
méthode de modèle.

4

Si les canards ne sont pas triés, ils sont échangés avec la méthode concrète `swap()` dans `Arrays` :

`échanger()`

Tableaux

`trier()`
`échanger()`

5

La méthode `sort()` continue de comparer et d'échanger les canards jusqu'à ce que le tableau soit dans le bon ordre !

there are no Dumb Questions

Q:

Est-ce vraiment le modèle de méthode de modèle, ou essayez-vous trop fort ?

UN:

Le modèle nécessite l'implémentation d'un algorithme et la fourniture de l'implémentation des étapes par des sous-classes, ce qui n'est clairement pas le cas de la fonction Arrays sort() ! Mais, comme nous le savons, les modèles courants ne sont pas toujours identiques à ceux des manuels. Ils doivent être modifiés pour s'adapter au contexte et aux contraintes d'implémentation.

Les concepteurs de la méthode Arrays sort() avaient quelques contraintes. En général, vous ne pouvez pas sous-classer un tableau Java et ils voulaient que le tri soit utilisé sur tous les tableaux (et chaque tableau est une classe différente). Ils ont donc défini une méthode statique et ont différé la partie comparaison de l'algorithme aux éléments à trier.

Ainsi, même s'il ne s'agit pas d'une méthode de modèle classique, cette implémentation reste dans l'esprit du modèle de méthode de modèle. De plus, en éliminant l'obligation de sous-classer les tableaux pour utiliser cet algorithme, ils ont rendu le tri plus flexible et plus utile à certains égards.

Q:

Cette implémentation du tri ressemble en fait plus au modèle de stratégie qu'au modèle de méthode de modèle. Pourquoi le considérons-nous comme une méthode de modèle ?

UN:

Vous pensez probablement que c'est parce que le modèle Strategy utilise la composition d'objets. Vous avez raison dans un sens : nous utilisons l'objet Arrays pour trier notre tableau, ce qui est similaire à Strategy. Mais rappelez-vous que dans Strategy, la classe avec laquelle vous composez implémente l'algorithme entier. L'algorithme implémenté par Arrays pour sort() est incomplet ; il a besoin d'une classe pour compléter la méthode compareTo() manquante. Donc, de cette façon, cela ressemble davantage à Template Method.

Q:

Existe-t-il d'autres exemples de méthodes de modèle dans Java API ?

UN:

Oui, vous les trouverez à plusieurs endroits. Par exemple, java.io possède une méthode read() dans InputStream que les sous-classes doivent implémenter et qui est utilisée par la méthode de modèle read(byte b[], int off, int len).



Nous savons que nous devrions privilégier la composition à l'héritage, n'est-ce pas ? Eh bien, les implémenteurs de la méthode de modèle sort() ont décidé de ne pas utiliser l'héritage et d'implémenter sort() comme une méthode statique composée avec un Comparable au moment de l'exécution. En quoi est-ce mieux ? En quoi est-ce pire ? Comment aborderiez-vous ce problème ? Les tableaux Java rendent-ils cela particulièrement délicat ?



Pensez à un autre modèle qui est une spécialisation de la méthode template. Dans cette spécialisation, des opérations primitives sont utilisées pour créer et renvoyer des objets. De quel modèle s'agit-il ?

Swingin' avec des cadres

Prochainement dans notre safari de méthodes de modèles... gardez un œil sur les JFrames swinguants !

Si vous n'avez jamais rencontré JFrame, c'est le conteneur Swing le plus basique et il hérite d'une méthode paint(). Par défaut, paint() ne fait rien car c'est un hook ! En remplaçant paint(), vous pouvez vous insérer dans l'algorithme de JFrame pour afficher sa zone de l'écran et avoir votre propre sortie graphique incorporée dans le JFrame. Voici un exemple d'une simplicité embarrassante d'utilisation d'un JFrame pour remplacer la méthode de hook paint() :



```
classe publique MyFrame étend JFrame {
```

```
    public MyFrame(Chaîne titre) {
        super(titre);
        ceci.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Nous étendons JFrame, qui contient une méthode update() qui contrôle l'algorithme de mise à jour de l'écran. Nous pouvons nous connecter à cet algorithme en remplaçant la méthode de hook paint().

```
        ceci.setSize(300,300);
        ceci.setVisible(true);
```

Ne regardez pas derrière le rideau ! Juste quelques initialisations ici...

```
}
```

L'algorithme de mise à jour de JFrame appelle paint(). Par défaut, paint() ne dessine rien... c'est un hook. Nous remplaçons paint() et demandons à JFrame de dessiner un message dans la fenêtre.

```
public void paint(Graphiques graphiques) {
```

```
    super.paint(graphiques);
    Chaîne msg = "Je suis le roi !!";
    graphics.drawString(msg, 100, 100);
```

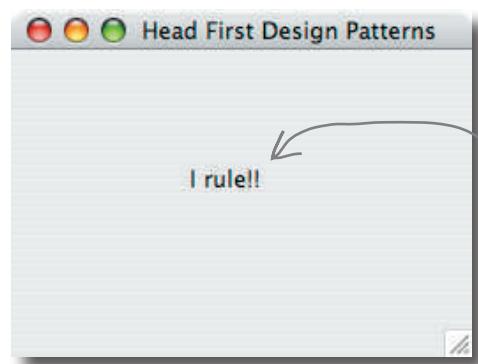
```
}
```

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

```
    MyFrame myFrame = new MyFrame("Modèles de conception Head First");
```

```
}
```

```
}
```



Voici le message qui est peint dans le cadre parce que nous nous sommes connectés à la méthode paint().



Listes personnalisées avec AbstractList

Notre dernière étape du safari : AbstractList.

Les collections de listes en Java, comme ArrayList et LinkedList, étendent la classe AbstractList, qui fournit certaines des implémentations de base du comportement des listes. Si vous souhaitez créer votre propre liste personnalisée (par exemple, une liste contenant uniquement des chaînes), vous pouvez le faire en étendant AbstractList afin d'obtenir gratuitement ce comportement de liste de base.

AbstractList possède une méthode de modèle, subList(), qui s'appuie sur deux méthodes abstraites, get() et size(). Ainsi, lorsque vous étendez AbstractList pour créer votre propre liste personnalisée, vous fournirez des implémentations pour ces méthodes.

Voici une implémentation d'une liste personnalisée qui contient uniquement des objets String et utilise des tableaux pour l'implémentation sous-jacente :

```
classe publique MyStringList étend AbstractList<String> {
    privé String[] maListe;
    MyStringList(String[] chaînes) {
        maListe = chaînes;
    }
    chaîne publique get(int index) {
        retourner maListe[index];
    }
    public int taille() {
        renvoie maListe.length;
    }
    public String set(int index, String élément) {
        Chaîne oldString = maListe[index];
        maListe[index] = élément;
        retourner ancienneChaîne;
    }
}
```



obtenir(3);
taille();

Nous créons une liste personnalisée en étendant AbstractList.

Nous devons implémenter les méthodes get() et size(), qui sont toutes deux utilisées par la méthode modèle subList().

Nous implémentons également une méthode set() afin de pouvoir modifier la liste.

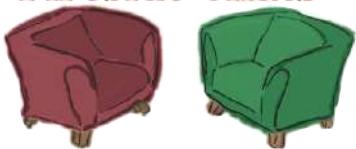
Testez la méthode de modèle subList() dans votre implémentation MyStringList comme ceci :

```
String[] ducks = { "Canard colvert", "Canard roux", "Canard en caoutchouc", "Canard leurre" };
MyStringList ducksList = new MyStringList(canards);
Liste ducksSubList = ducksList.subList(2, 3);
```

Créez une sous-liste d'un élément commençant à l'index 2... le canard en caoutchouc, bien sûr.



Fireside Chats



La conférence de ce soir :**Méthode de modèle et méthode de comparaison de stratégie.**

Méthode de modèle :

Hé Stratégie, qu'est-ce que tu fais dans mon chapitre ? Je pensais que je me retrouverais coincé avec quelqu'un d'ennuyeux comme Factory Method.

Je plaisantais ! Mais sérieusement, qu'est-ce que tu fais ici ?
On n'a pas eu de tes nouvelles depuis sept chapitres !

Vous devriez peut-être rappeler au lecteur de quoi vous parlez, car cela fait si longtemps.

Hé, cela ressemble beaucoup à ce que je fais. Mais mon intention est un peu différente de la vôtre ; mon travail consiste à définir les grandes lignes d'un algorithme, mais à laisser mes sous-classes faire une partie du travail. De cette façon, je peux avoir différentes implémentations des étapes individuelles d'un algorithme, tout en gardant le contrôle sur la structure de l'algorithme. Il semble que vous deviez abandonner le contrôle de vos algorithmes.

Stratégie:



Non, c'est moi, mais soyez prudent : vous et Factory Method êtes liés, n'est-ce pas ?

J'ai entendu dire que tu étais sur la dernière version de ton chapitre et j'ai pensé que je passerais voir comment ça se passait. Nous avons beaucoup de points communs, alors j'ai pensé que je pourrais peut-être t'aider...

Je ne sais pas, depuis le chapitre 1, les gens m'arrêtent dans la rue en me disant : « N'êtes-vous pas ce modèle... ? » Je pense donc qu'ils savent qui je suis. Mais pour votre bien : je définis une famille d'algorithmes et je les rends interchangeables. Comme chaque algorithme est encapsulé, le client peut facilement utiliser différents algorithmes.

Je ne suis pas sûr que je le dirais tout à fait comme *ça que...* et de toute façon, je ne suis pas obligé d'utiliser l'héritage pour les implémentations d'algorithmes. J'offre aux clients un choix d'implémentation d'algorithmes via la composition d'objets.

Méthode de modèle :

Je m'en souviens. Mais j'ai plus de contrôle sur mon algorithme et je ne duplique pas le code. En fait, si chaque partie de mon algorithme est la même à l'exception, disons, d'une ligne, alors mes classes sont beaucoup plus efficaces que les vôtres. Tout mon code dupliqué est placé dans la superclasse, de sorte que toutes les sous-classes peuvent le partager.

Ouais, eh bien, je suis rée*je* suis content pour toi, mais n'oublie pas que je suis le modèle le plus utilisé. Pourquoi ? Parce que je fournis une méthode fondamentale pour la réutilisation du code qui permet aux sous-classes de spécifier le comportement. Je suis sûr que tu peux voir que c'est parfait pour créer des frameworks.

Comment ça ? Ma superclasse est abstraite.

Comme je l'ai dit, la stratégie, je suis rée*je* suis content pour toi. Merci d'être passé, mais je dois terminer le reste de ce chapitre.

Je l'ai compris. Ne le fais pasAppeler-nous, nous vous appellerons...

Stratégie:

Vous pourriez être un peu plus efficace (juste un peu) et avoir besoin de moins d'objets.*En*ous pourriez aussi être un peu moins compliqué par rapport à mon modèle de délégation, mais je suis plus flexible car j'utilise la composition d'objets. Avec moi, les clients peuvent modifier leurs algorithmes au moment de l'exécution simplement en utilisant un objet de stratégie différent. Allez, ils ne m'ont pas choisi pour le chapitre 1 pour rien !

Ouais, je suppose... mais qu'en est-il de la dépendance ? Tu es bien plus dépendante que moi.

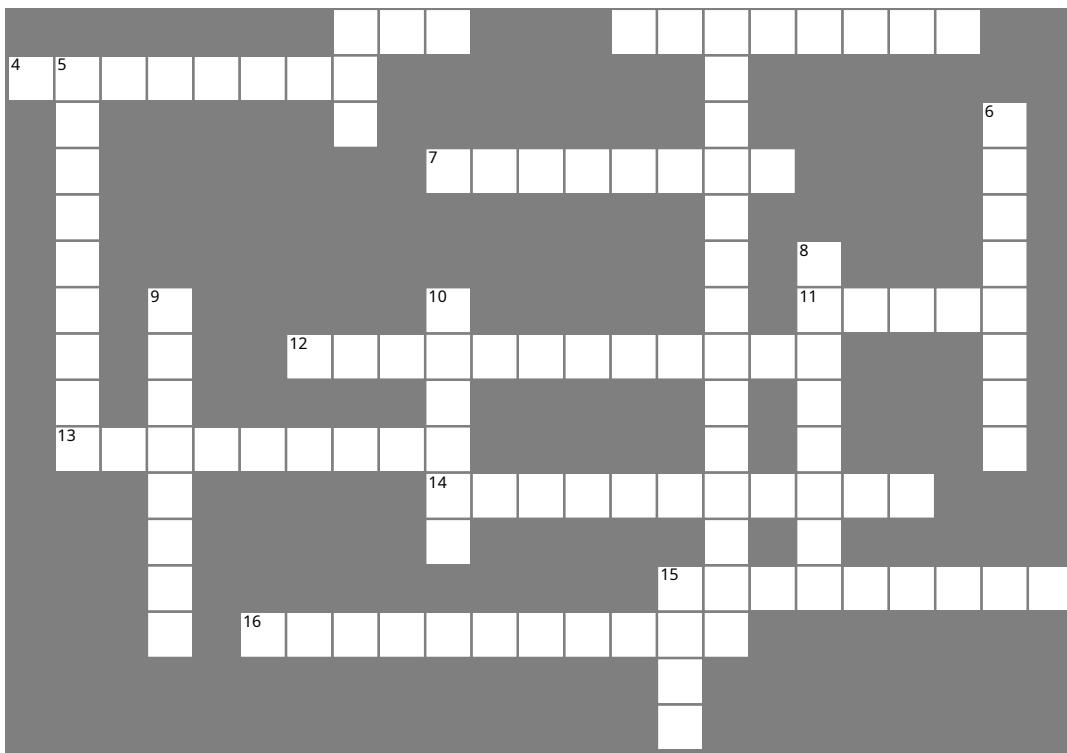
Mais vous devez dépendre des méthodes implémentées dans vos sous-classes, qui font partie de votre algorithme. Je ne dépend pas de personne ; je peux faire tout l'algorithme moi-même !

Ok, ok, ne sois pas trop sensible. Je te laisse travailler, mais fais-moi savoir si tu as besoin de mes techniques spéciales de toute façon ; je suis toujours ravi de t'aider.



Mots croisés sur les modèles de conception

C'est à nouveau ce moment...



À TRAVERS

1. Riri, Loulou et Dewey pèsent tous _____ livres.
 2. La méthode de modèle est généralement définie dans une classe _____.
 3. Dans ce chapitre, nous vous avons donné plus de _____.
 4. Les étapes de l'algorithme qui doivent être fournies par les sous-classes sont généralement déclarées _____.
 5. La méthode hook JFrame que nous avons remplacée pour imprimer « Je règne !! »
 6. _____ a une méthode de modèle subList().
 7. Type de tri utilisé dans les tableaux.
 8. Le modèle de méthode de modèle utilise _____ pour différer l'implémentation à d'autres classes.
 9. « Ne nous appelez pas, nous vous appellerons » est connu sous le nom de principe _____.

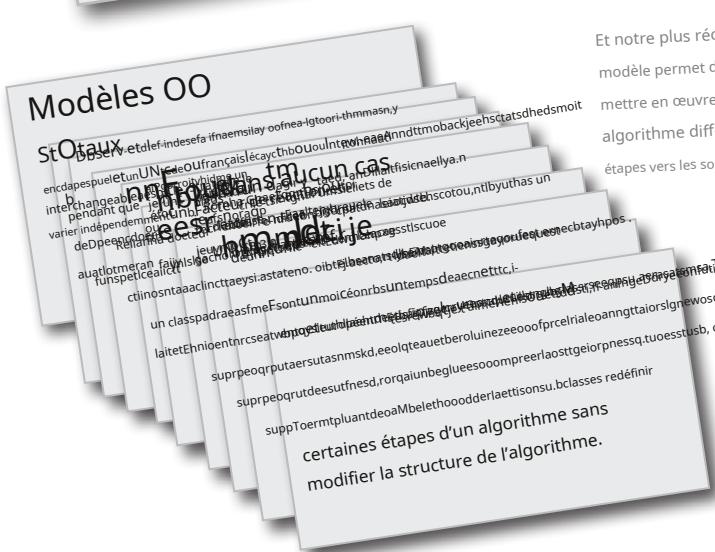
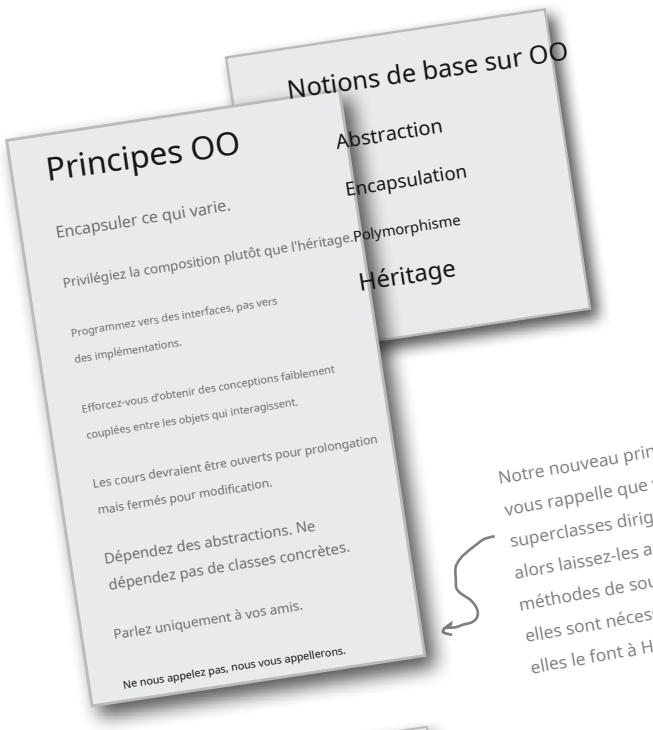
VERS I F BAS

1. Café et ____.
 3. La méthode Factory est une _____ de la méthode Template.
 5. Une méthode de modèle définit les étapes d'un _____.
6. Modèle à grosse tête.
 8. Les étapes de l'algorithme _____ sont implémentées par des méthodes hook.
 9. Notre café préféré à Objectville.
 10. La classe Arrays implémente sa méthode de modèle en tant que méthode _____.
15. Une méthode dans la superclasse abstraite qui ne fait rien ou fournit un comportement par défaut est appelée une méthode _____.



Des outils pour votre boîte à outils de conception

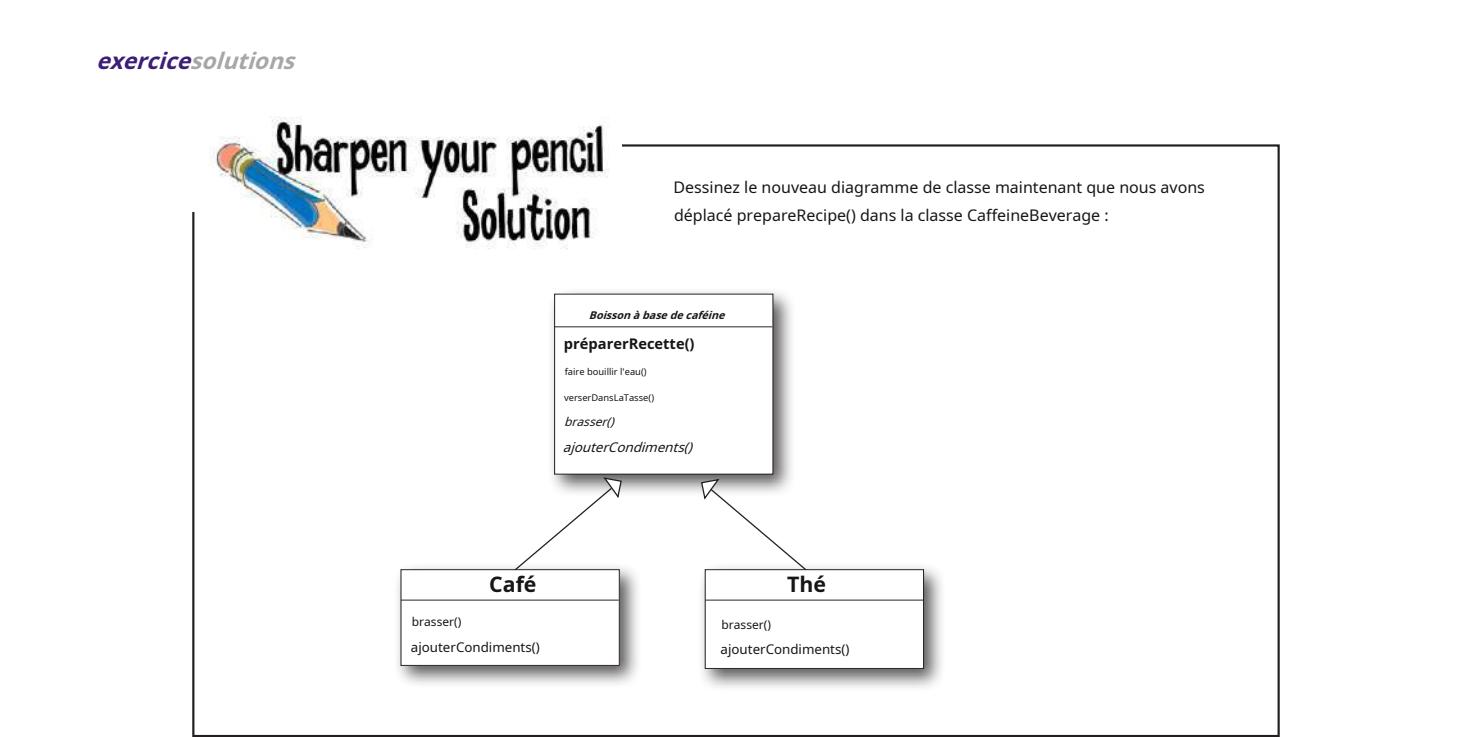
Nous avons ajouté Template Method à votre boîte à outils. Avec Template Method, vous pouvez réutiliser le code comme un pro tout en gardant le contrôle de vos algorithmes.



BULLET POINTS

- f Une méthode de modèle définit les étapes d'un algorithme, en s'adressant à des sous-classes pour la mise en œuvre de ces étapes.
- f Le modèle de méthode de modèle nous offre une technique importante pour la réutilisation du code.
- f La classe abstraite de la méthode modèle peut définir des méthodes concrètes, des méthodes abstraites et des hooks.
- f Les méthodes abstraites sont implémenté par des sous-classes.
- f Les hooks sont des méthodes qui ne font rien ou qui ont un comportement par défaut dans la classe abstraite, mais qui peuvent être remplacées dans la sous-classe.
- f Pour empêcher les sous-classes de modifier l'algorithme dans la méthode modèle, déclarez la méthode modèle comme finale.
- f Le principe d'Hollywood nous guide pour placer la prise de décision dans des modules de haut niveau qui peuvent décider comment et quand appeler les modules de bas niveau.
- f Vous verrez de nombreuses utilisations du modèle de méthode de modèle dans le code du monde réel, mais (comme pour tout modèle) ne vous attendez pas à ce que tout soit conçu « selon les règles ».
- f Les modèles de stratégie et de méthode de modèle encapsulent tous deux des algorithmes, le premier par composition et l'autre par héritage.
- f La méthode Factory est une spécialisation de la méthode Template.

Sharpen your pencil Solution



Associez chaque motif à sa description :

Modèle

Description

Méthode de modèle

Encapsulez les comportements interchangeables et utilisez la délégation pour décider quel comportement utiliser.

Stratégie

Les sous-classes décident comment implémenter les étapes d'un algorithme.

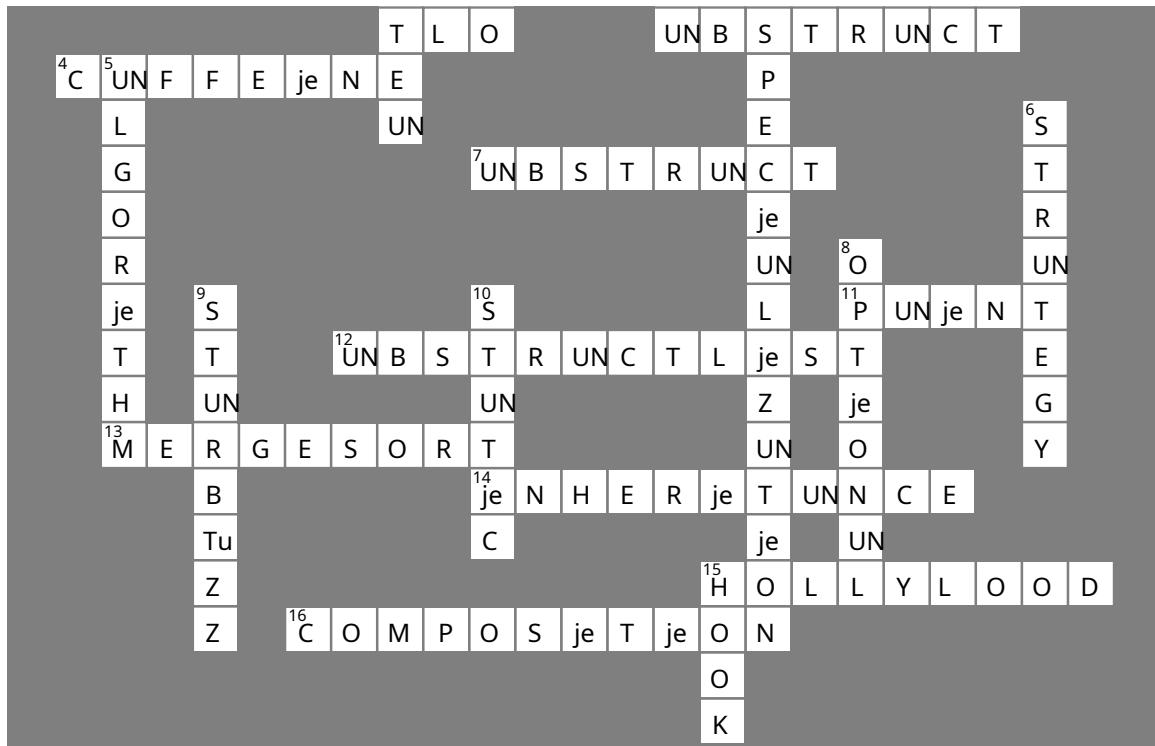
Méthode d'usine

Les sous-classes décident quelles classes concrètes créer.



Solution de mots croisés sur les modèles de conception

C'est à nouveau ce moment...



des modèles Iterator et Composite

Bien géré *Collections*

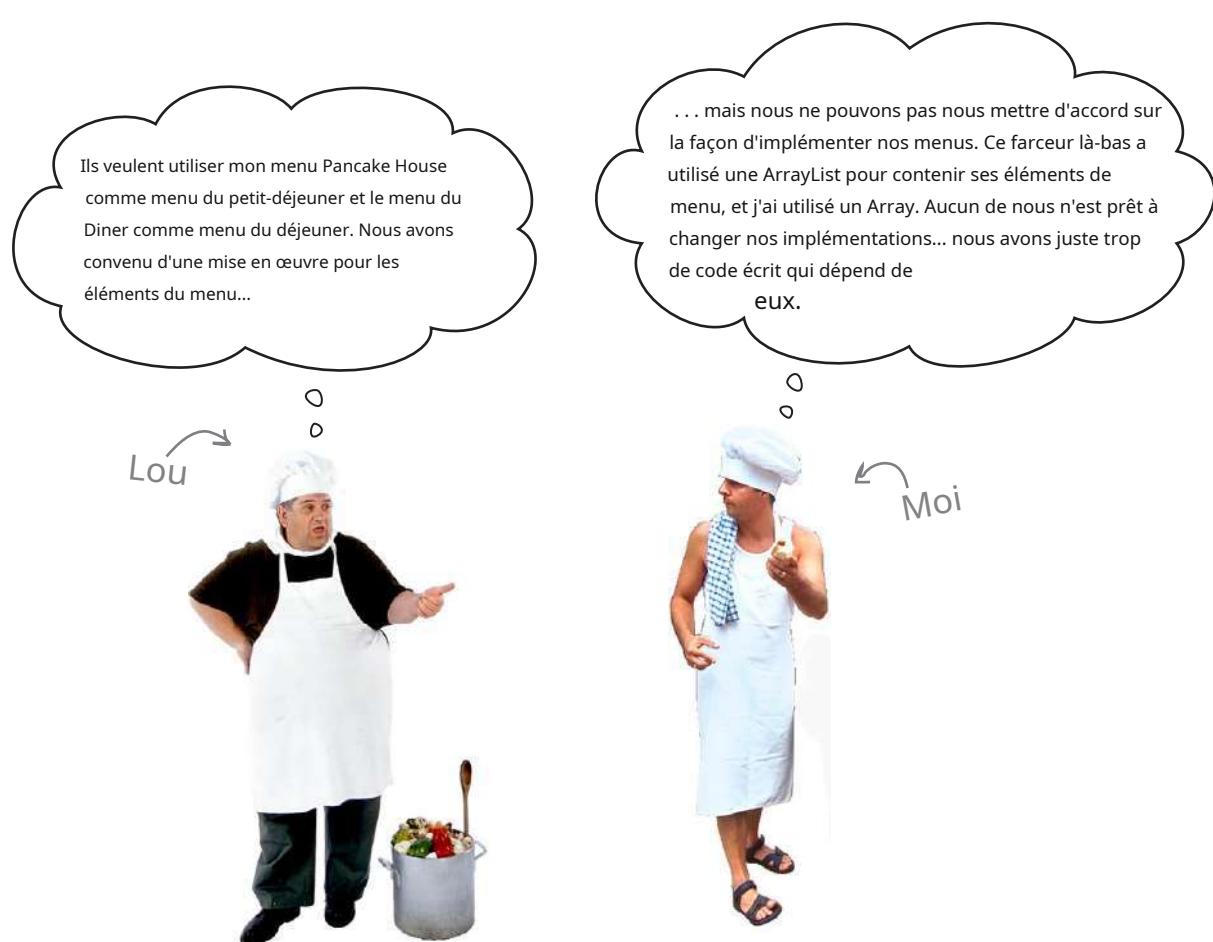


Il existe de nombreuses façons d'intégrer des objets dans une collection.

Placez-les dans un tableau, une pile, une liste, une table de hachage, faites votre choix. Chacun a ses propres avantages et inconvénients. Mais à un moment donné, vos clients voudront parcourir ces objets, et quand ils le feront, allez-vous leur montrer votre implémentation ? Nous espérons certainement que non ! Ce ne serait pas professionnel. Eh bien, vous n'avez pas à risquer votre carrière ; dans ce chapitre, vous allez voir comment vous pouvez permettre à vos clients de parcourir vos objets sans jamais avoir un aperçu de la façon dont vous les stockez. Vous allez également apprendre à créer des super collections d'objets qui peuvent sauter par-dessus des structures de données impressionnantes en une seule fois. Et si cela ne suffit pas, vous allez également apprendre une chose ou deux sur la responsabilité des objets.

Dernières nouvelles : Objectville Diner et Objectville Pancake House fusionnent

C'est une excellente nouvelle ! Nous pouvons désormais déguster ces délicieux petits-déjeuners à base de crêpes au Pancake House et ces délicieux déjeuners au Diner, le tout au même endroit. Mais il semble y avoir un léger problème...



Découvrez les éléments du menu

Au moins, Lou et Mel sont d'accord sur l'implémentation des éléments de menu. Examinons les éléments de chaque menu et examinons également l'implémentation.

Le menu du Diner propose de nombreux plats pour le déjeuner, tandis que celui du Pancake House propose des plats pour le petit-déjeuner. Chaque plat du menu possède un nom, une description et un prix.

classe publique MenuItem {

Nom de la chaîne ;

Description de la chaîne ;

booléen végétarien;

double prix;

```
public MenuItem(Chaîne nom,
                Description de la chaîne,
                booléen végétarien,
                (prix double)
{
    ceci.nom = nom;
    this.description = description;
    this.vegetarian = végétarien; this.price
    = prix;
}
```

chaîne publique getName() {

renvoyer le nom;

}

chaîne publique getDescription() {

description du retour;

}

public double getPrice() {

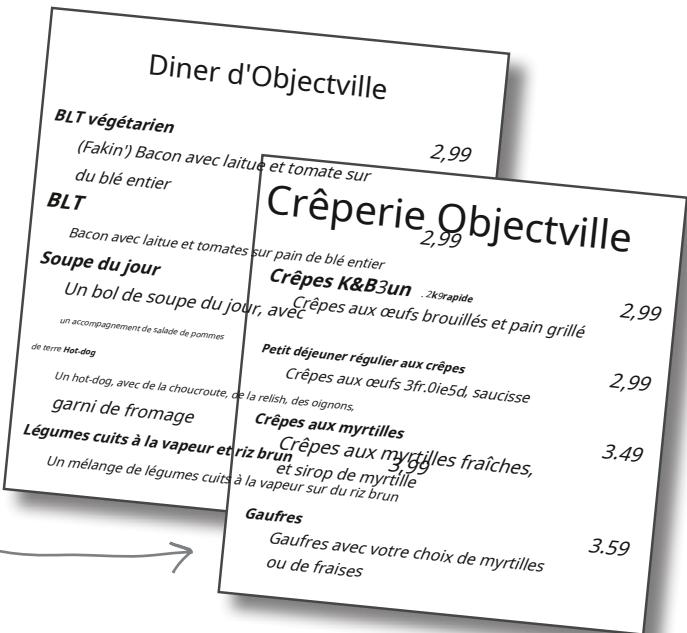
prix de retour;

}

public boolean isVegetarian() {

retour végétarien;

}



Un élément de menu se compose d'un nom, d'une description, d'un indicateur pour indiquer si l'élément est végétarien et d'un prix. Vous transmettez toutes ces valeurs au constructeur pour initialiser l'élément de menu.

Ces méthodes getter vous permettent d'accéder aux champs de l'élément de menu.

Implémentations du menu de Lou et Mel

Voyons maintenant ce sur quoi Lou et Mel se disputent. Ils ont tous les deux beaucoup de temps et de code investis dans la façon dont ils stockent leurs éléments de menu dans un menu, et beaucoup d'autres codes qui en dépendent.

Voici la mise en œuvre du menu Pancake House par Lou.

```

classe publique PancakeHouseMenu {
    Liste<MenuItem> éléments de menu ;

    MenuPancakeHouse public() {
        menuItems = nouvelle ArrayList<MenuItem>();

        addItem("Petit déjeuner aux crêpes de K&B",
            "Crêpes aux œufs brouillés et pain grillé", c'est vrai,
            2,99);

        addItem("Petit déjeuner aux crêpes classique",
            "Crêpes aux œufs au plat, saucisse", faux,
            2,99);

        addItem("Crêpes aux myrtilles",
            "Crêpes aux myrtilles fraîches", c'est vrai,
            3,49);

        addItem("Gaufres",
            "Gaufres avec votre choix de myrtilles ou de fraises", c'est vrai,
            3,59);
    }

    public void addItem(Chaîne nom, Chaîne description,
        booléen végétarien, prix double)
    {
        MenuItem menuItem = new MenuItem(nom, description, végétarien, prix);
        menuItems.add(menuItem);
    }

    public ArrayList<MenuItem> getMenuItems() {
        retourner les éléments de menu;
    }

    // autres méthodes de menu ici
}

```

J'ai utilisé une ArrayList pour pouvoir facilement l'étendre mon menu.



Lou utilise une classe ArrayList pour stocker ses éléments de menu.

Chaque élément de menu est ajouté à l'ArrayList ici, dans le constructeur.

Chaque élément de menu possède un nom, une description, s'il s'agit ou non d'un élément végétarien et le prix.

Pour ajouter un élément de menu, Lou crée un nouvel objet MenuItem, en passant chaque argument, puis l'ajoute à ArrayList.

La méthode getMenuItems() renvoie la liste des éléments de menu.

Lou a un tas d'autres codes de menu qui dépendent de l'implémentation d'ArrayList. Il ne veut pas avoir à réécrire tout ce code !



Haah ! Une ArrayList... J'ai utilisé un VRAI tableau pour pouvoir contrôler la taille maximale de mon menu.

```

classe publique DinerMenu {
    statique final int MAX_ITEMS = 6; int
    numberofItems = 0;
    MenuItem[] éléments de menu;

public DinerMenu() {
    menuItems = nouveau MenuItem[MAX_ITEMS];
    addItem("BLT végétarien",
        "(Fakin') Bacon avec laitue et tomate sur du blé entier", true, 2.99); addItem("BLT",
        "Bacon avec laitue et tomate sur blé entier", false, 2.99); addItem("Soupe
        du jour",
        "Soupe du jour, accompagnée d'une salade de pommes de terre", false, 3.29);
    addItem("Hotdog",
        « Un hot-dog, avec de la choucroute, de la relish, des oignons, garni de fromage »,
        faux, 3.05);
    // quelques autres éléments du menu du restaurant ont été ajoutés ici
}

public void addItem(Chaîne nom, Chaîne description,
booléen végétarien, prix double)
{
    MenuItem menuItem = new MenuItem(nom, description, végétarien, prix); si (nombre
    d'éléments >= MAX_ITEMS) {
        System.err.println("Désolé, le menu est plein ! Impossible d'ajouter un élément au
        menu"); } else {
        menuItems[nombred'éléments] = menuItem;
        nombred'éléments = nombred'éléments + 1;
    }
}

public MenuItem[] getMenuItems()
{
    retourner les éléments de menu;
}

// autres méthodes de menu ici

```

Et voici l'implémentation du menu Diner par Mel.

Mel adopte une approche différente ; il utilise une classe Array pour pouvoir contrôler la taille maximale du menu.

Comme Lou, Mel crée ses éléments de menu dans le constructeur, en utilisant la méthode d'assistance addItem().

addItem() prend tous les paramètres nécessaires pour créer un MenuItem et en instancie un. Il vérifie également que nous n'avons pas atteint la limite de taille du menu.

Mel souhaite spécifiquement garder son menu en dessous d'une certaine taille (probablement pour ne pas avoir à se souvenir de trop de recettes).

getMenuItems() renvoie le tableau des éléments de menu.

Comme Lou, Mel a un tas de code qui dépend de l'implémentation de son menu sous forme de tableau. Il est trop occupé à cuisiner pour réécrire tout ça.

Quel est le problème d'avoir deux représentations de menu différentes ?

Pour comprendre pourquoi avoir deux représentations de menu différentes complique les choses, essayons d'implémenter un client qui utilise les deux menus. Imaginez que vous ayez été embauché par la nouvelle société issue de la fusion du Diner et du Pancake House pour créer une serveuse compatible Java (il s'agit d'Objectville, après tout). La spécification de la serveuse compatible Java précise qu'elle peut imprimer un menu personnalisé pour les clients à la demande, et même vous dire si un élément du menu est végétarien sans avoir à demander au cuisinier : c'est une innovation !

Voyons les spécifications de la serveuse, puis voyons ce qu'il faudrait pour la mettre en œuvre...

Spécification de la serveuse compatible Java

Serveuse compatible Java : nom de code « Alice »

imprimerMenu()
- imprime chaque élément des menus du petit-déjeuner et du déjeuner

imprimerBreakfastMenu()
- imprime uniquement des articles pour le petit-déjeuner

imprimerMenuDéjeuner()
- imprime uniquement les articles du déjeuner

imprimerMenuVégétarien()
- imprime tous les éléments du menu végétarien

isItemVegetarian(nom)
- étant donné le nom d'un élément, renvoie vrai si l'élément est végétarien, sinon, renvoie faux



↑
La spécification pour la serveuse

Implémentation de la spécification : notre première tentative

Commençons par expliquer comment nous implémenterions la méthode printMenu() :

- Pour imprimer tous les éléments de chaque menu, vous devez appeler la méthode getMenuItems() sur PancakeHouseMenu et DinerMenu pour récupérer leurs éléments de menu respectifs. Notez que chacun renvoie un type différent :

```
PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
ArrayList<MenuItem> breakfastItems = pancakeHouseMenu.getMenuItems();
```

```
MenuDiner MenuDinerMenu = new DinerMenu(); MenuItem[]
```

La méthode ressemble pareil, mais les appels reviennent différents types.

L'implémentation est visible : les éléments du petit-déjeuner sont dans une ArrayList et les éléments du déjeuner sont dans un tableau.

- Maintenant, pour imprimer les éléments du PancakeHouseMenu, nous allons parcourir les éléments de la liste de tableaux breakfastItems. Et pour imprimer les éléments du Diner, nous allons parcourir le tableau.

```
pour (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = breakfastItems.get(i);
    System.out.print(menuItem.getName() + " ");
    System.out.println(menuItem.getPrice() + " ");
    System.out.println(menuItem.getDescription());
}

pour (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
    System.out.print(menuItem.getName() + " ");
    System.out.println(menuItem.getPrice() + " ");
    System.out.println(menuItem.getDescription());
}
```

Maintenant, nous devons mettre en œuvre deux différentes boucles à parcourir les deux implémentations de la éléments de menu...

... une boucle pour l'ArrayList...

... et un autre pour le tableau.

- L'implémentation de toutes les autres méthodes dans Waitress sera une variante de ce thème. Nous aurons toujours besoin d'obtenir les deux menus et d'utiliser deux boucles pour parcourir leurs éléments. Si un autre restaurant avec une implémentation différente est acquis, nous aurons trois boucles.



Sur la base de notre implémentation de printMenu(), lesquelles des affirmations suivantes s'appliquent ?

- A. Nous codons pour les implémentations concrètes de PancakeHouseMenu et DinerMenu, pas pour une interface.
- B. La serveuse n'implémente pas l'API Java Waitress et elle n'est donc pas adhérant à une norme.
- C. Si nous décidions de passer de l'utilisation de DinerMenu à un autre type de menu qui implémentait sa liste d'éléments de menu avec une table de hachage, nous devrions modifier beaucoup de code dans Waitress.
- D. La serveuse doit savoir comment chaque menu représente sa collection interne d'éléments de menu ; cela viole l'encapsulation.
- E. Nous avons un code en double : la méthode printMenu() nécessite deux boucles distinctes pour itérer sur les deux types de menus différents. Et si nous ajoutions un troisième menu, nous aurions encore une autre boucle.
- F. L'implémentation n'est pas basée sur MXML (Menu XML) et n'est donc pas aussi interopérable qu'elle devrait l'être.

Et maintenant ?

Mel et Lou nous mettent dans une position difficile. Ils ne veulent pas changer leurs implémentations car cela impliquerait de réécrire beaucoup de code dans chaque classe de menu respective. Mais si l'un d'eux ne cède pas, nous allons devoir implémenter une Waitress qui sera difficile à maintenir et à étendre.

Ce serait vraiment sympa si nous pouvions trouver un moyen de leur permettre d'implémenter la même interface pour leurs menus (ils sont déjà proches, sauf pour le type de retour de la méthode getMenuItems()). De cette façon, nous pouvons minimiser les références concrètes dans le code Waitress et, espérons-le, également nous débarrasser des multiples boucles nécessaires pour parcourir les deux menus.

Ça a l'air bien ? Et bien, comment allons-nous faire ça ?

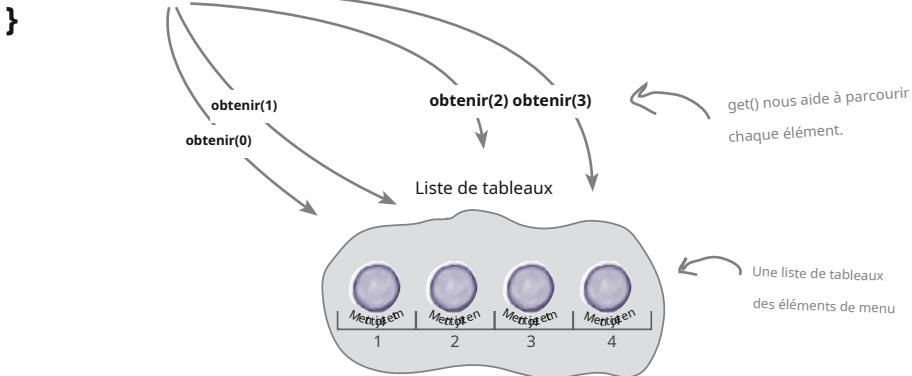
Pouvons-nous encapsuler l'itération ?

Si nous avons appris une chose dans ce livre, c'est qu'il faut encapsuler ce qui varie. Ce qui change ici est évident : l'itération causée par différentes collections d'objets renvoyées par les menus. Mais pouvons-nous encapsuler cela ? Examinons cette idée...

- 1 Pour parcourir les éléments du petit-déjeuner, nous utilisons les méthodes size() et get() sur ArrayList :

```
pour (int i = 0; i < breakfastItems.size(); i++) {
```

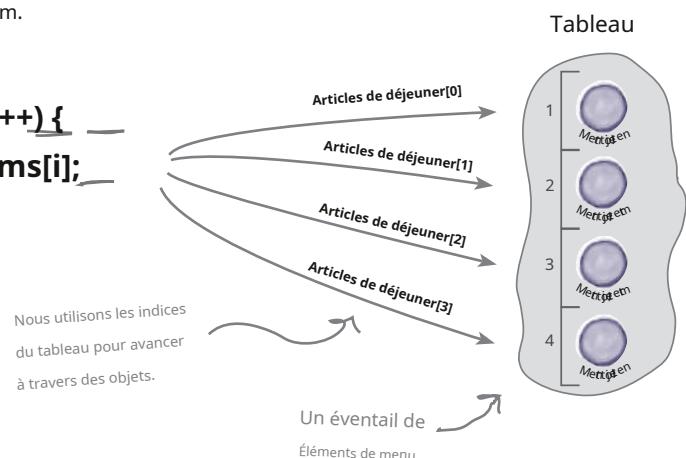
```
MenuItem menuItem = breakfastItems.get(i);
```



- 2 Et pour parcourir les éléments du déjeuner, nous utilisons le champ de longueur du tableau et la notation d'indice de tableau sur le tableau MenuItem.

```
pour (int i = 0; i < lunchItems.length; i++) {
```

```
MenuItem menuItem = lunchItems[i];
```



- 3 Et maintenant, que se passe-t-il si nous créons un objet, appelons-le un Iterator, qui encapsule la façon dont nous parcourons une collection d'objets ? Essayons ceci sur ArrayList :

Nous demandons au breakfastMenu un itérateur de ses MenuItem.

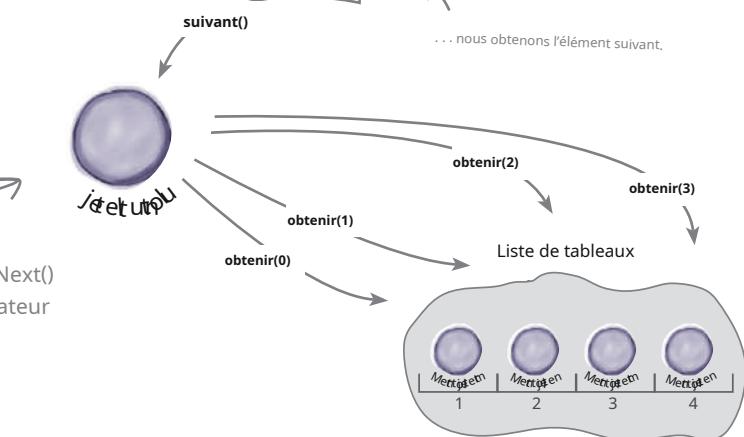
Itérateur itérateur = breakfastMenu.createIterator();

```
tandis que (iterator.hasNext()) {
    MenuItem menuItem = iterator.next();
}
```

Le client appelle simplement hasNext() et next(); dans les coulisses, l'itérateur appelle get() sur ArrayList.

Et bien qu'il reste encore d'autres articles...

... nous obtenons l'élément suivant.



- 4 Essayons cela également sur le tableau :

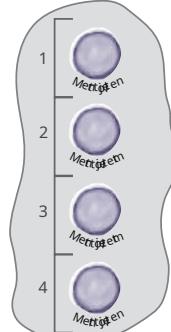
Itérateur itérateur = lunchMenu.createIterator();

```
tandis que (iterator.hasNext()) {
    MenuItem menuItem = iterator.next();
}
```

Waouh, ce code est exactement le même que le menu du petit déjeuner code.

Même situation ici : le client appelle simplement hasNext() et next(); en coulisses, l'itérateur indexe dans le tableau.

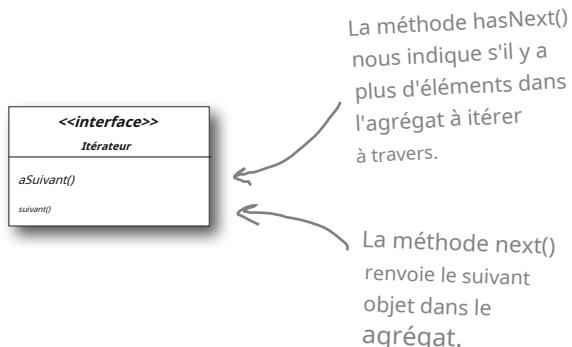
Tableau



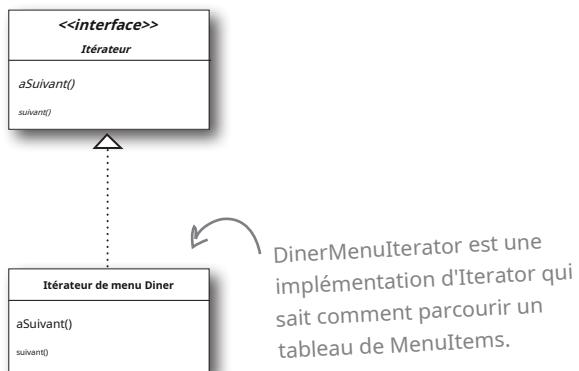
Découvrez le modèle Iterator

Eh bien, il semble que notre plan d'encapsulation de l'itération pourrait bien fonctionner ; et comme vous l'avez probablement déjà deviné, il s'agit d'un modèle de conception appelé modèle d'itérateur.

La première chose que vous devez savoir sur le modèle Iterator est qu'il repose sur une interface appelée Iterator. Voici une interface Iterator possible :



Maintenant, une fois que nous avons cette interface, nous pouvons implémenter des itérateurs pour tout type de collection d'objets : tableaux, listes, tables de hachage... choisissez votre collection d'objets préférée. Disons que nous voulions implémenter l'itérateur pour le tableau utilisé dans le DinerMenu. Cela ressemblerait à ceci :



Allons-y et implémentons cet itérateur et incorporons-le dans DinerMenu pour voir comment cela fonctionne...



Ajout d'un itérateur à DinerMenu

Pour ajouter un itérateur au DinerMenu, nous devons d'abord définir l'interface Iterator :

```
interface publique Itérateur {  
    booléen hasNext();  
    Élément de menu suivant();  
}
```

Voici nos deux méthodes :

La méthode hasNext() renvoie un booléen indiquant s'il y a ou non d'autres éléments sur lesquels itérer...

... et la méthode next() renvoie l'élément suivant.

Et maintenant, nous devons implémenter un itérateur concret qui fonctionne pour le menu Diner :

```
classe publique DinerMenuIterator implémente Iterator {  
    MenuItem[] éléments;  
    int position = 0;  
  
    public DinerMenuIterator(MenuItem[] éléments) {  
        this.items = éléments;  
    }
```

Nous mettons en œuvre le Interface d'itérateur.

position maintient la position actuelle de l'itération sur le tableau.

Le constructeur prend le tableau d'éléments de menu sur lesquels nous allons parcourir.

```
élément de menu public suivant() {  
    MenuItem menuItem = éléments[position];  
    position = position + 1;  
    retourner menuItem;  
}
```

La méthode next() renvoie l'élément suivant du tableau et incrémente la position.

```
public booléen hasNext() {  
    si (position >= éléments.length || éléments[position] == null) {  
        retourner faux;  
    } autre {  
        retourne vrai ;  
    }  
}
```

La méthode hasNext() vérifie si nous avons vu tous les éléments du tableau et renvoie true s'il y en a plus à parcourir.

Etant donné que le chef du restaurant a alloué un tableau de taille maximale, nous devons vérifier non seulement si nous sommes à la fin du tableau, mais également si l'élément suivant est nul, ce qui indique qu'il n'y a plus d'éléments.

Retravailler le DinerMenu avec Iterator

Ok, nous avons l'itérateur. Il est temps de l'intégrer dans le DinerMenu ; tout ce que nous devons faire est d'ajouter une méthode pour créer un DinerMenuIterator et le renvoyer au client :

```

classe publique DinerMenu {
    statique final int MAX_ITEMS = 6; int
    numberOfItems = 0;
    MenuItem[] éléments de menu;

    // constructeur ici

    // ajouter un élément ici

    public MenuItem[] getMenuItems() {
        returner les éléments de menu;
    }

    Itérateur public createIterator() {
        renvoyer un nouveau DinerMenuIterator(menuItems);
    }

    // autres méthodes de menu ici
}

```

Nous n'aurons plus besoin de la méthode `getMenuItems()` ; en fait, nous n'en voulons pas car elle expose notre implémentation interne !

Voici la méthode `createIterator()`. Elle crée un `DinerMenuIterator` à partir du tableau `menuItems` et le renvoie au client.

Nous renvoyons l'interface `Iterator`. Le client n'a pas besoin de savoir comment les `MenuItem`s sont maintenus dans le `DinerMenu`, ni comment le `DinerMenuIterator` est implémenté. Il a juste besoin d'utiliser les itérateurs pour parcourir les éléments du menu.



Allez-y et implémentez vous-même le `PancakeHouseIterator` et apportez les modifications nécessaires pour l'intégrer dans `PancakeHouseMenu`.

Correction du code de la serveuse

Nous devons maintenant intégrer le code de l'itérateur dans la classe `Waitress`. Nous devrions pouvoir nous débarrasser d'une partie de la redondance du processus. L'intégration est assez simple : nous créons d'abord une méthode `printMenu()` qui prend un `Iterator` ; puis nous utilisons la méthode `createIterator()` sur chaque menu pour récupérer l'`Iterator` et le transmettre à la nouvelle méthode.



```

classe publique Serveuse {
    MenuCrêpes et CrêpesMenuCrêpes et CrêpesMenu;
    MenuDinerMenuDinerMenu;

    Serveuse publique (PancakeHouseMenu pancakeHouseMenu, DinerMenu dinerMenu) {
        ceci.pancakeHouseMenu = pancakeHouseMenu;
        ceci.dinerMenu = dinerMenu;
    }

    public void printMenu() {
        Itérateur pancakeIterator = pancakeHouseMenu.createIterator(); Itérateur
        dinerIterator = dinerMenu.createIterator();

        System.out.println("MENU\n----\nPETIT DEJEUNER");
        printMenu(pancakeIterator);
        System.out.println("\nDÉJEUNER");
        printMenu(dinerIterator);
    }

    privé void printMenu(Iterator itérateur) {
        tandis que (itérateur.hasNext()) {
            MenuItem menuItem = itérateur.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }

    // autres méthodes ici
}

```

Dans le constructeur, la classe `Waitress` prend les deux menus.

Le `printMenu()` la méthode crée maintenant deux itérateurs, un pour chaque menu...

... puis appelle la fonction `printMenu()` surchargée à chaque itérateur.

Testez s'il y a d'autres éléments. Obtenez le élément suivant.

Le surchargé `imprimerMenu()` méthode utilise l'**itérateur** à passer à travers les éléments du menu et les imprimer.

Notez que nous n'avons plus qu'une seule boucle.

Utilisez l'objet pour obtenir le nom, le prix, et description et les imprimer.

Tester notre code

Il est temps de tout mettre à l'épreuve. Écrivons un code de test et voyons comment fonctionne la serveuse...

```
classe publique MenuTestDrive {
    public static void main(String args[]) {
        MenuCrêpesMaison pancakeHouseMenu = new MenuCrêpesMaison();
        MenuDiner menuDinerMenu = new MenuDiner();

        Serveuse serveuse = nouvelle Serveuse(pancakeHouseMenu, dinerMenu);

        serveuse.printMenu();
    }
}
```

Nous créons d'abord les nouveaux menus.

Ensuite, nous créons une serveuse et passons voici les menus.

Ensuite, nous les imprimons.

Voici le test...

```
Aide sur la fenêtre d'édition de fichier
% java DinerMenuTestDrive
MENU
-----
PETIT-DÉJEUNER
Petit-déjeuner aux crêpes de K&B, 2,99 $ - Crêpes avec œufs brouillés et pain grillé
Petit-déjeuner aux crêpes ordinaire, 2,99 $ - Crêpes avec œufs au plat et saucisses
Crêpes aux myrtilles, 3,49 $ - Crêpes faites avec des myrtilles fraîches Gaufres, 3,59
$ - Gaufres avec votre choix de myrtilles ou de fraises

DÉJEUNER
BLT végétarien, 2,99 $ - (Fakin') Bacon avec laitue et tomate sur du blé entier BLT, 2,99
$ - Bacon avec laitue et tomate sur du blé entier
Soupe du jour, 3.29 -- Soupe du jour, accompagnée d'une salade de pommes de terre
Hot Dog, 3,05 $ - Un hot dog avec de la choucroute, de la relish, des oignons, garni de fromage.
Légumes cuits à la vapeur et riz brun, 3,99 $ - Légumes cuits à la vapeur sur du riz brun. Pâtes,
3,89 $ - Spaghetti à la sauce marinara et une tranche de pain au levain.

%
```

Nous commençons par itérer à travers la carte des crêpes...

... et puis le déjeuner menu, tout avec le même itération code.

Qu'avons-nous fait jusqu'à présent ?

Pour commencer, nous avons rendu nos cuisiniers d'Objectville très heureux. Ils ont réglé leurs différends et conservé leurs propres implémentations. Une fois que nous leur avons donné un PancakeHouseMenuIterator et un DinerMenuIterator, tout ce qu'ils avaient à faire était d'ajouter une méthode `createIterator()` et ils avaient terminé.

Nous nous sommes également aidés dans ce processus. La Waitress sera beaucoup plus facile à entretenir et à étendre à l'avenir. Examinons exactement ce que nous avons fait et réfléchissons aux conséquences :



Difficile à entretenir

Mise en œuvre de la serveuse

Les menus ne sont pas bien encapsulés ; nous pouvons voir que le Diner utilise un `ArrayList` et le Pancake House un `Array`.

Nous avons besoin de deux boucles pour parcourir les `MenuItem`s.

La serveuse est liée à des classes concrètes (`MenuItem[]` et `ArrayList`).

La serveuse est liée à deux classes de menu concrètes différentes, bien que leurs interfaces soient presque identiques.

Nouveau, branché

Serveuse Propulsé par Iterator

Les implémentations de menu sont désormais encapsulées. La serveuse n'a aucune idée de la manière dont les menus contiennent leur collection d'éléments de menu.

Tout ce dont nous avons besoin est une boucle qui gère de manière polymorphe n'importe quelle collection d'éléments à condition qu'elle implémente `Iterator`.

La serveuse utilise désormais une interface (`Iterator`).

Les interfaces de menu sont désormais exactement les mêmes et, oh oh, nous n'avons toujours pas d'interface commune, ce qui signifie que la serveuse est toujours liée à deux classes de menu concrètes. Nous ferions mieux de régler ce problème.

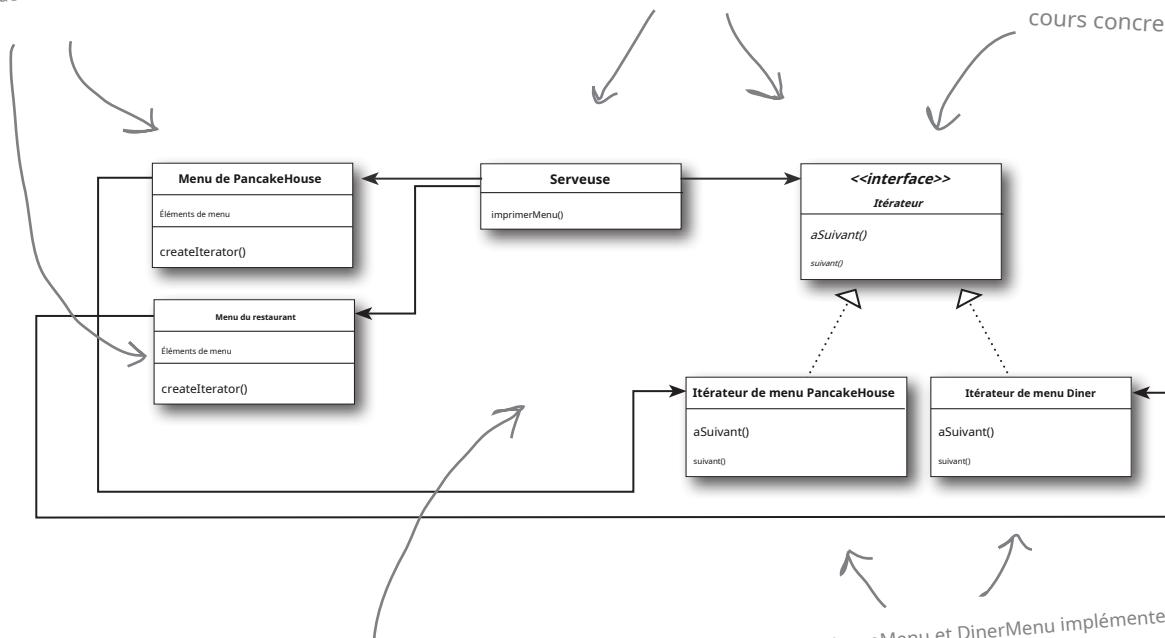
Révision de notre conception actuelle...

Avant de nettoyer les choses, jetons un œil à notre conception actuelle.

Ces deux menus implémentent exactement le même ensemble de méthodes, mais ils n'implémentent pas la même interface. Nous allons résoudre ce problème et libérer la serveuse de toute dépendance vis-à-vis de menus concrets.

L'Iterator permet à la Serveuse d'être découpée de l'implémentation réelle des classes concrètes. Elle n'a pas besoin de savoir si un Menu est implémenté avec un Array, une ArrayList ou des Post-it. Tout ce qui l'intéresse, c'est qu'elle puisse faire en sorte qu'un Iterator effectue son itération.

Nous utilisons maintenant un itérateur commun interface et nous avons mis en œuvre deux cours concrets.



Notez que l'itérateur nous donne un moyen de parcourir les éléments d'un agrégat sans forcer l'agrégat à encombrer sa propre interface avec un tas de méthodes pour prendre en charge la traversée de ses éléments. Il permet également à l'implémentation de l'itérateur de vivre en dehors de l'agrégat ; en d'autres termes, nous avons encapsulé l'itération.

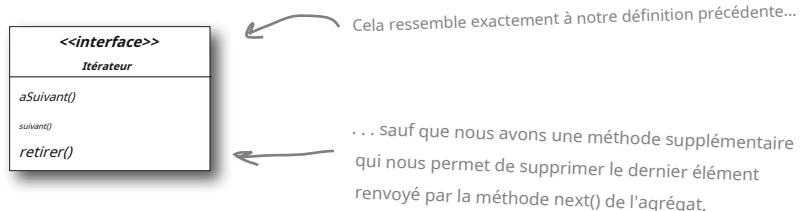
PancakeHouseMenu et DinerMenu implémentent la nouvelle méthode `createIterator()` ; ils sont responsables de la création de l'itérateur pour les implémentations de leurs éléments de menu respectifs.

Apporter quelques améliorations...

Ok, nous savons que les interfaces de PancakeHouseMenu et DinerMenu sont exactement les mêmes et pourtant nous n'avons pas défini d'interface commune pour elles. Nous allons donc faire cela et nettoyer un peu plus la Waitress.

Vous vous demandez peut-être pourquoi nous n'utilisons pas l'interface Java Iterator. Nous l'avons fait pour que vous puissiez voir comment créer un itérateur à partir de zéro. Maintenant que nous avons fait cela, nous allons passer à l'utilisation de l'interface Java Iterator, car nous obtiendrons un effet de levier important en l'implémentant au lieu de notre interface Iterator développée en interne. Quel type d'effet de levier ? Vous le verrez bientôt.

Commençons par examiner l'interface java.util.Iterator :



Cela va être un jeu d'enfant : nous devons juste changer l'interface que PancakeHouseMenuIterator et DinerMenuIterator étendent, n'est-ce pas ? Presque... en fait, c'est même plus simple que ça. Non seulement java.util a sa propre interface Iterator, mais ArrayList a une méthode iterator() qui renvoie un itérateur. En d'autres termes, nous n'avons jamais eu besoin d'implémenter notre propre itérateur pour ArrayList. Cependant, nous aurons toujours besoin de notre implémentation pour DinerMenu car elle repose sur un Array, qui ne prend pas en charge la méthode iterator().

there are no
Dumb Questions

Q: Que faire si je ne souhaite pas offrir la possibilité de supprimer quelque chose de la collection d'objets sous-jacente ?

UN: La méthode remove() est considérée comme facultative. Vous n'êtes pas obligé de fournir la fonctionnalité remove. Mais vous devez fournir la méthode car elle fait partie de l'interface Iterator. Si vous n'autorisez pas remove() dans votre itérateur, vous souhaiterez lever l'exception d'exécution java.lang.UnsupportedOperationException. La documentation de l'API Iterator spécifie que cette exception peut être levée à partir de remove() et que tout client qui est un bon citoyen recherchera cette exception lors de l'appel de la méthode remove().

Q: Comment remove() se comporte-t-il sous plusieurs threads qui peuvent utiliser différents itérateurs sur la même collection d'objets ?

UN: Le comportement de la méthode remove() n'est pas spécifié si la collection change pendant que vous l'exécutez. Vous devez donc être prudent lors de la conception de votre propre code multithread lorsque vous accédez simultanément à une collection.

Nettoyer les choses avec java.util.Iterator

Commençons par le PancakeHouseMenu. Le remplacer par java.util.Iterator va être facile. Il suffit de supprimer la classe PancakeHouseMenuIterator, d'ajouter une importation java.util.Iterator en haut de PancakeHouseMenu et de modifier une ligne du PancakeHouseMenu :

```
public Iterator<MenuItem> createIterator() {
    renvoie menuItems.iterator();
}
```



Au lieu de créer notre propre itérateur maintenant, nous appelons simplement la méthode iterator() sur la liste de tableaux menuItems (plus d'informations à ce sujet dans un instant).

Et voilà, PancakeHouseMenu est terminé.

Nous devons maintenant effectuer les modifications pour permettre à DinerMenu de fonctionner avec java.util.Iterator.

```
importer java.util.Iterator;
```



Nous importons d'abord java.util.Iterator, l'interface que nous allons implémenter.

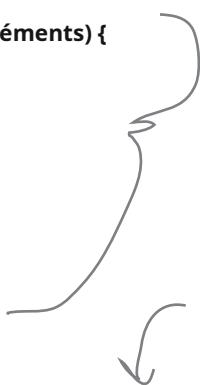
```
classe publique DinerMenuIterator implémente Iterator<MenuItem> {
    MenuItem[] éléments;
    int position = 0;

    public DinerMenuIterator(MenuItem[] éléments) {
        this.items = éléments;
    }

    élément de menu public suivant() {
        //implémentation ici
    }

    public booléen hasNext() {
        //implémentation ici
    }

    public void remove() {
        générer une nouvelle UnsupportedOperationException
        (« Vous ne devriez pas essayer de supprimer des éléments de menu. »)
    }
}
```



Aucun de nos changements d'implémentation actuels...

N'oubliez pas que la méthode remove() est facultative dans l'interface Iterator. Demander à notre serveuse de supprimer des éléments de menu n'a pas vraiment de sens, nous lancerons donc simplement une exception si elle essaie.

Nous y sommes presque...

Il ne nous reste plus qu'à donner aux menus une interface commune et à retravailler un peu la serveuse. L'interface du menu est assez simple : nous voudrons peut-être y ajouter quelques méthodes supplémentaires par la suite, comme addItem(), mais pour l'instant nous laisserons les chefs contrôler leurs menus en gardant cette méthode hors de l'interface publique :

```
interface publique Menu {  
    public Iterator<MenuItem> createIterator();  
}
```

Il s'agit d'une interface simple qui permet simplement aux clients d'obtenir un itérateur pour les éléments du menu.

Maintenant, nous devons ajouter unmet en œuvre Menuaux définitions de classe PancakeHouseMenu et DinerMenu et mettre à jour la classe Waitress :

```
importer java.util.Iterator;
```

Maintenant, la serveuse utilise également java.util.Iterator.

```
classe publique Serveuse {  
    Menu Menu de la maison des crêpes;  
    Menu Menu du restaurant;  
  
    serveuse publique( MenuMenu de la crêperie, Menu(menu du dîner) {  
        ceci.pancakeHouseMenu = pancakeHouseMenu;  
        ceci.dinerMenu = dinerMenu;  
    }  
  
    public void printMenu() {  
        Itérateur <élément de menu> dinnerIterator = dinnerMenu.createIterator();  
        Itérateur <élément de menu>  
        System.out.println("MENU\n---\nPETIT DEJEUNER");  
        printMenu(pancakeIterator);  
        System.out.println("\nDÉJEUNER");  
        printMenu(dinnerIterator);  
    }  
  
    privé void printMenu(Iterator itérateur) {  
        tandis que (iterator.hasNext()) {  
            MenuItem menuItem = iterator.next();  
            System.out.print(menuItem.getName() + ", ");  
            System.out.print(menuItem.getPrice() + " -- ");  
            System.out.println(menuItem.getDescription());  
        }  
    }  
    // autres méthodes ici  
}
```

Nous devons remplacer les classes de menu concrètes par l'interface de menu.

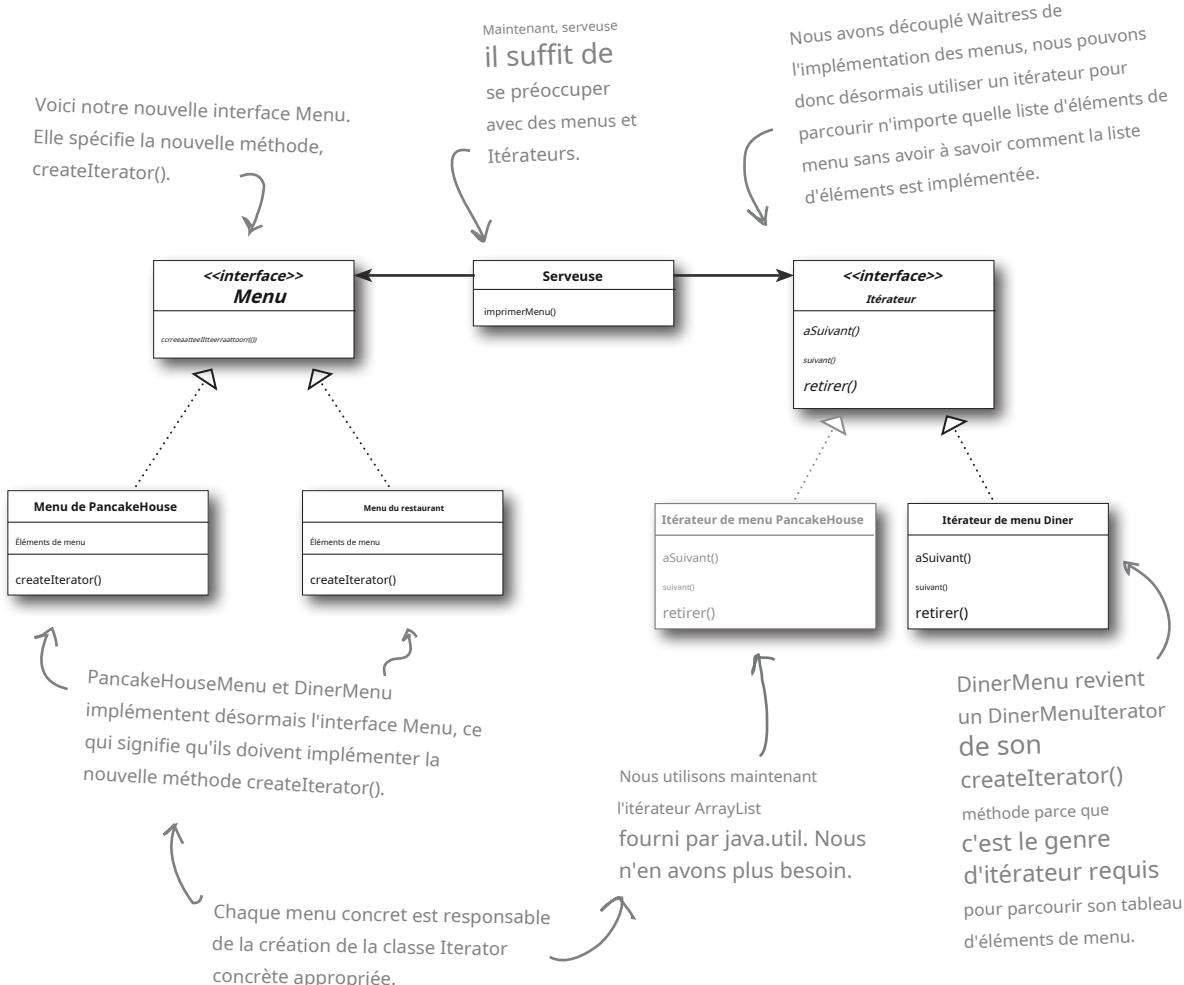
Rien ne change ici.

Qu'est-ce que cela nous apporte ?

Les classes PancakeHouseMenu et DinerMenu implémentent une interface, Menu. Cela permet à la serveuse de faire référence à chaque objet de menu en utilisant l'interface plutôt que la classe concrète. Nous réduisons donc la dépendance entre la serveuse et les classes concrètes en « programmant sur une interface, pas une implémentation ».

De plus, la nouvelle interface Menu possède une méthode, `createIterator()`, qui est implémentée par PancakeHouseMenu et DinerMenu. Chaque classe de menu assume la responsabilité de créer un itérateur concret adapté à son implémentation interne des éléments de menu.

Cela résout le problème de la serveuse en fonction de la Menus concrets.



Modèle d'itérateur défini

Vous avez déjà vu comment implémenter le modèle Iterator avec votre propre itérateur. Vous avez également vu comment Java prend en charge les itérateurs dans certaines de ses classes orientées collection (ArrayList). Il est maintenant temps de vérifier la définition officielle du modèle :

Le modèle d'itérateur fournit un moyen d'accéder aux éléments d'un objet agrégé de manière séquentielle sans exposer sa représentation sous-jacente.

Cela a beaucoup de sens : le modèle vous donne un moyen de parcourir les éléments d'un agrégat sans avoir à savoir comment les choses sont représentées sous les couvertures. Vous avez vu cela avec les deux implémentations de Menus. Mais l'effet de l'utilisation d'itérateurs dans votre conception est tout aussi important : une fois que vous avez un moyen uniforme d'accéder aux éléments de tous vos objets d'agrégat, vous pouvez écrire du code polymorphe qui fonctionne avec *n'importe lequel* de ces agrégats, tout comme la méthode printMenu(), qui ne se soucie pas de savoir si les éléments de menu sont contenus dans un tableau ou une liste de tableaux (ou tout autre élément pouvant créer un itérateur), tant qu'elle peut mettre la main sur un itérateur.

L'autre impact important sur votre conception est que le modèle Iterator prend la responsabilité de parcourir les éléments et donne cette responsabilité à l'objet itérateur, et non à l'objet agrégé. Cela permet non seulement de simplifier l'interface et l'implémentation de l'agrégat, mais également de retirer à l'agrégat la responsabilité de l'itération et de garder l'agrégat concentré sur les éléments sur lesquels il devrait se concentrer (gérer une collection d'objets), et non sur l'itération.

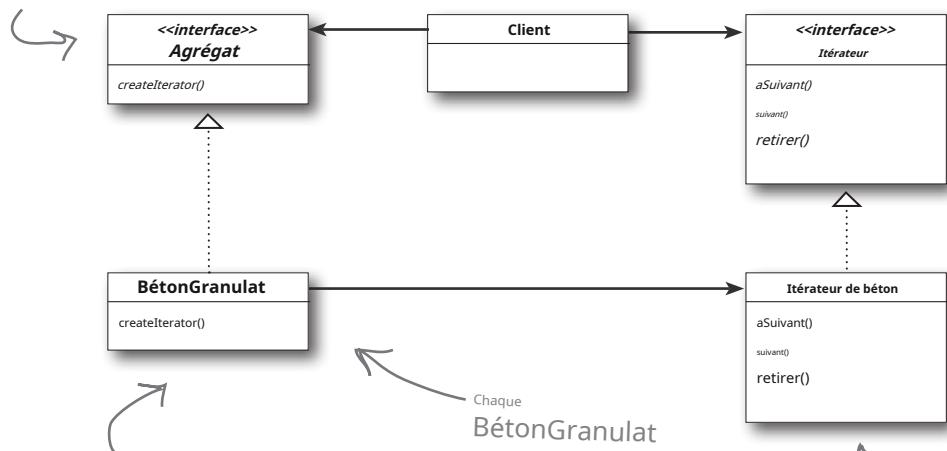
Le modèle d'itérateur permet de parcourir les éléments d'un agrégat sans exposer le implémentation sous-jacente.

Il place également la tâche de traversée sur l'objet itérateur, et non dans l'ensemble, ce qui simplifie le interface et implémentation agrégées, et place la responsabilité là où elle devrait être.

La structure du modèle d'itérateur

Regardons le diagramme de classe pour mettre tous les éléments en contexte...

Avoir une interface commune pour vos agrégats est pratique pour votre client ; cela dissocie votre client de l'implémentation de votre collection d'objets.



Le bétonagrégat possède une collection d'objets et d'outils la méthode qui renvoie un itérateur pour sa collection.

Chaque BétonGranulat est responsable de instantiation d'un ConcreteIterator qui peut parcourir sa collection d'objets.

L'interface Iterator fournit l'interface que tous les itérateurs doit mettre en œuvre, et un ensemble de méthodes permettant de parcourir éléments d'une collection. Ici, nous utilisons java.util.Iterator. Si vous ne voulez pas utiliser l'itérateur de Java interface, vous pouvez Créez toujours le vôtre.

Le ConcreteIterator est responsable de la gestion de la position actuelle de l'itération.

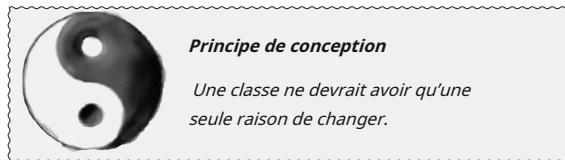


Le diagramme de classe du modèle Iterator ressemble beaucoup à un autre modèle que vous avez étudié ; pouvez-vous imaginer de quoi il s'agit ? Astuce : une sous-classe décide quel objet créer.

Le principe de responsabilité unique

Quoiqu'en avons permis à nos agrégats d'implémenter leurs collections internes et les opérations associées ET les méthodes d'itération ? Eh bien, nous savons déjà que cela augmenterait le nombre de méthodes dans l'agrégat, mais alors ? Pourquoi est-ce si mauvais ?

Eh bien, pour comprendre pourquoi, vous devez d'abord reconnaître que lorsque nous permettons à une classe non seulement de s'occuper de ses propres affaires (gérer une sorte d'agrégat) mais aussi d'assumer davantage de responsabilités (comme l'itération), nous avons alors donné à la classe deux raisons de changer. Deux ? Oui, deux : elle peut changer si la collection change d'une manière ou d'une autre, et elle peut changer si la façon dont nous itérons change. Donc, une fois de plus, notre ami CHANGE est au centre d'un autre principe de conception :



Nous savons que nous voulons éviter tout changement dans nos classes, car la modification du code offre toutes sortes d'opportunités pour que des problèmes s'installent. Avoir deux façons de changer augmente la probabilité que la classe change à l'avenir, et lorsque cela se produit, cela va affecter deux aspects de votre conception.

La solution ? Le principe nous guide pour attribuer chaque responsabilité à une classe, et à une seule classe.

C'est vrai, c'est aussi simple que ça, et puis ça ne l'est pas : séparer les responsabilités dans la conception est l'une des choses les plus difficiles à faire. Notre cerveau est tout simplement trop doué pour voir un ensemble de comportements et les regrouper même lorsqu'il y a en fait deux ou plusieurs responsabilités. La seule façon de réussir est d'examiner attentivement vos conceptions et de surveiller les signaux indiquant qu'une classe change de plusieurs manières à mesure que votre système se développe.

Chaque responsabilité d'une classe est un domaine de changement potentiel. plus d'une responsabilité implique plus d'un domaine de changement.

Ce principe nous guide pour confier à chaque classe une seule responsabilité.



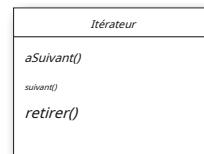
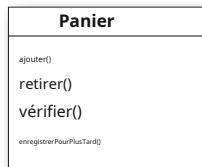
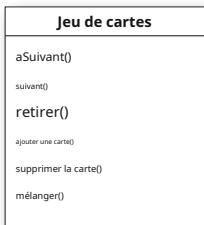
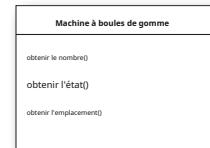
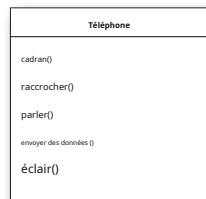
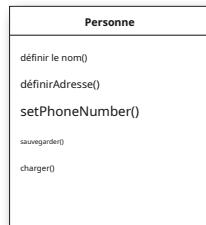
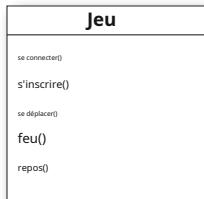
Cohésion est un terme que vous entendrez utilisé pour mesurer la mesure dans laquelle une classe ou un module soutient un objectif ou une responsabilité unique.

On dit qu'un module ou la classe a *une haute cohésion* lorsqu'il est conçu autour d'un ensemble de fonctions connexes, et nous disons qu'il a *une faible cohésion* lorsqu'il est conçu autour d'un ensemble de fonctions sans rapport entre elles.

La cohésion est un concept plus général que le principe de responsabilité unique, mais les deux sont étroitement liés. Les classes qui adhèrent à ce principe ont tendance à avoir une forte cohésion et sont plus faciles à maintenir que les classes qui assument de multiples responsabilités et ont une faible cohésion.



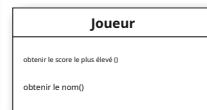
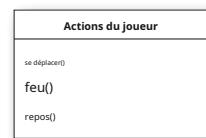
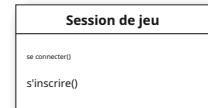
Examinez ces classes et déterminez lesquelles ont plusieurs responsabilités.



Zone de casque de sécurité. Attention aux hypothèses qui tombent



Déterminez si ces classes ont une cohésion faible ou élevée.



there are no Dumb Questions

Q:

J'ai vu d'autres livres montrer le diagramme de classe Iterator avec les méthodes `first()`, `next()`, `isDone()` et `currentItem()`. Pourquoi ces méthodes sont-elles différentes ?

UN:

Ce sont les noms de méthodes « classiques » qui ont été utilisés. Ces noms ont changé au fil du temps et nous avons maintenant `next()`, `hasNext()` et même `remove()` dans `java.util.Iterator`.

Regardons les méthodes classiques. Les méthodes `next()` et `currentItem()` ont été fusionnées en une seule méthode dans `java.util`. La méthode `isDone()` est devenue `hasNext()`, mais nous n'avons pas de méthode correspondant à `first()`. C'est parce qu'en Java, nous avons tendance à obtenir simplement un nouvel itérateur chaque fois que nous devons recommencer la traversée. Néanmoins, vous pouvez voir qu'il y a très peu de différence entre ces interfaces. En fait, il existe toute une gamme de comportements que vous pouvez donner à vos itérateurs. La méthode `remove()` est un exemple d'extension dans `java.util.Iterator`.

Q:

J'ai entendu parler des itérateurs « internes » et des itérateurs « externes ». Que sont-ils ? Quel type avons-nous implémenté dans l'exemple ?

UN:

Nous avons implémenté un itérateur externe, ce qui signifie que le client contrôle l'itération en appelant `next()` pour obtenir l'élément suivant. Un itérateur interne est contrôlé par l'itérateur lui-même. Dans ce cas, comme c'est l'itérateur qui parcourt les éléments, vous devez lui dire quoi faire avec ces éléments au fur et à mesure qu'il les parcourt. Cela signifie que vous avez besoin d'un moyen de transmettre une opération à un itérateur. Les itérateurs internes sont moins flexibles que les itérateurs externes, car le client n'a pas le contrôle de l'itération. Cependant, certains pourraient affirmer qu'ils sont plus faciles à utiliser, car il suffit de

Donnez-leur une opération et dites-leur d'itérer, et ils feront tout le travail pour vous.

Q:

Puis-je implémenter un itérateur qui peut aussi bien reculer qu'avancer ?

UN:

Probablement. Dans ce cas, vous souhaiterez probablement ajouter deux méthodes, une pour accéder à l'élément précédent et une pour vous indiquer quand vous êtes au début de la collection d'éléments. Le framework Collection de Java fournit un autre type d'interface d'itérateur appelé `ListIterator`. Cet itérateur ajoute `previous()` et quelques autres méthodes à l'interface `Iterator` standard. Il est pris en charge par toute collection qui implémente l'interface `List`.

Q:

Qui définit l'ordre de l'itération dans une collection comme `Hashtable`, qui est intrinsèquement non ordonnée ?

UN:

Les itérateurs n'impliquent aucun ordre. Les collections sous-jacentes peuvent ne pas être ordonnées comme dans une table de hachage ou dans un sac ; elles peuvent même contenir des doublons. L'ordre est donc lié à la fois aux propriétés de la collection sous-jacente et à l'implémentation. En général, vous ne devez faire aucune supposition sur l'ordre, sauf si la documentation de la collection indique le contraire.

Q:

Vous avez dit que nous pouvons écrire du « code polymorphe » en utilisant un itérateur ; Tu expliques ça plus en détail ?

UN:

Lorsque nous écrivons des méthodes qui prennent des itérateurs comme paramètres, nous utilisons une itération polymorphe. Cela signifie que nous créons du code qui peut parcourir n'importe quelle collection tant qu'elle prend en charge `Iterator`. Peu importe la manière dont la collection est implémentée, nous pouvons toujours écrire du code pour parcourir celle-ci.

Q:

Si j'utilise Java, ne souhaiterai-je pas toujours utiliser l'interface `java.util.Iterator` afin de pouvoir utiliser mes propres implémentations d'itérateur avec des classes qui utilisent déjà les itérateurs Java ?

UN:

Probablement. Si vous disposez d'une interface `Iterator` commune, il vous sera certainement plus facile de mélanger et d'associer vos propres agrégats avec des agrégats Java comme `ArrayList` et `Vector`. Mais n'oubliez pas que si vous devez ajouter des fonctionnalités à votre interface `Iterator` pour vos agrégats, vous pouvez toujours étendre l'interface `Iterator`.

Q:

J'ai vu une interface d'énumération en Java ; est-ce que cela implémente le modèle `Iterator` ?

UN:

Nous en avons parlé dans le chapitre Adapter Pattern (Chapitre 7). Vous vous souvenez ? `java.util Enumeration` est une ancienne implémentation d'`Iterator` qui a depuis été remplacée par `java.util Iterator`. `Enumeration` possède deux méthodes, `hasMoreElements()`, correspondant à `hasNext()`, et `nextElement()`, correspondant à `next()`. Cependant, vous souhaiterez probablement utiliser `Iterator` plutôt qu'`Enumeration` car de plus en plus de classes Java le prennent en charge. Si vous devez passer de l'un à l'autre, consultez à nouveau le Chapitre 7 où vous avez implémenté l'adaptateur pour `Enumeration` et `Iterator`.

Q:

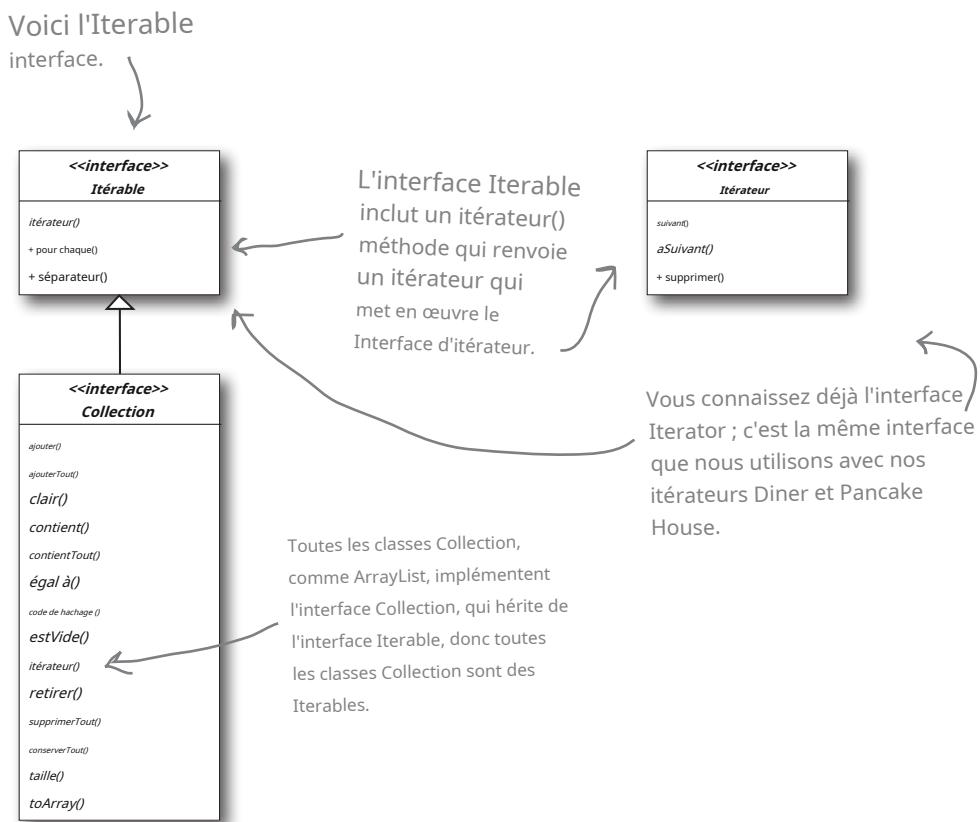
L'utilisation de la boucle `for` améliorée de Java est-elle liée aux itérateurs ?

UN:

Bonne question ! Et pour répondre à cette question, nous devons comprendre une autre interface, à savoir l'interface `Iterable` de Java. C'est le moment idéal pour le faire...

Découvrez l'interface Iterable de Java

Tu es déjà au courant de Java **Itérateur** interface, mais il y a une autre interface que vous devez respecter : **Iterable**. L'interface Iterable est implémentée par chaque type Collection en Java. Devinez quoi ? Dans votre code utilisant ArrayList, vous avez déjà utilisé cette interface. Jetons un œil à l'interface Iterable :



Si une classe implémente Iterable, nous savons que la classe implémente une méthode `iterator()`. Cette méthode renvoie un itérateur qui implémente l'interface Iterator. Cette interface inclut également une méthode `forEach()` par défaut qui peut être utilisée comme un autre moyen d'itérer dans la collection. En plus de tout cela, Java fournit même un sucre syntaxique intéressant pour l'itération, avec sa boucle for améliorée. Voyons comment cela fonctionne.

L'interface Iterable inclut également la méthode `spliterator()`, qui fournit des moyens encore plus avancés pour parcourir une collection.

Boucle for améliorée de Java

Prenons un objet dont la classe implémente l'interface Iterable... pourquoi pas la collection ArrayList que nous avons utilisée pour les éléments de menu de Pancake House :

```
Liste<MenuItem> menuItems = new ArrayList<MenuItem>();
```

Nous pouvons parcourir ArrayList comme nous l'avons fait :

```
Itérateur iterator = menu.iterator(); while  
(iterator.hasNext()) {  
    MenuItem menuItem = iterator.next();  
    System.out.print(menuItem.getName() + ", ");  
    System.out.print(menuItem.getPrice() + " -- ");  
    System.out.println(menuItem.getDescription());  
}
```



C'est ainsi que nous avons effectué l'itération sur nos collections, en utilisant un itérateur avec les méthodes hasNext() et next().

Ou, étant donné que nous savons qu'ArrayList est un Iterable, nous pourrions utiliser le raccourci amélioré de java :

```
pour (élément MenuItem : menu) {  
    System.out.print(menuItem.getName() + ", ");  
    System.out.print(menuItem.getPrice() + " -- ");  
    System.out.println(menuItem.getDescription());  
}
```



Ici, nous pouvons nous passer de l'itérateur explicite comme les méthodes hasNext() et next().



Passé rapide ; les tableaux ne sont pas des itérables

Nous avons de mauvaises nouvelles : le Diner n'a peut-être pas pris la meilleure décision en utilisant un tableau comme base pour ses menus. Il s'avère que les tableaux ne sont pas des collections Java et qu'ils n'implémentent donc pas l'interface Iterable. Étant donné cela, nous ne pouvons pas aussi facilement consolider notre code Waitress en une seule méthode qui prend un Iterable et l'utilise à la fois avec les breakfastItems du Pancake House et les lunchItems du Diner. Si vous essayez de modifier la méthode printMenu() de Waitress pour qu'elle prenne un Iterable au lieu d'un Iterator, et utilisez la boucle for-each au lieu de l'API Iterator, comme ceci :

```
public void printMenu(Iterable<MenuItem> itérable) {
    pour (MenuItem menuItem : itérable) {
        // imprimer l'élément de menu
    }
}
```

Cela ne fonctionnera que pour l'ArrayList que nous utilisons pour le menu Pancake House.



vous obtiendrez une erreur de compilation lorsque vous tenterez de transmettre le tableau lunchItems à printMenu() :

`printMenu(lunchItems);` ← Erreur de compilation ! Les tableaux ne sont pas des itérables.

car, encore une fois, les tableaux n'implémentent pas l'interface Iterable.

Si vous conservez les deux boucles dans le code de la serveuse, nous revenons à la case départ : la serveuse dépend à nouveau des types d'agrégrats que nous utilisons pour stocker les menus, et elle a un code en double : une boucle pour l'ArrayList et une boucle pour l'Array.

Alors, que faire ? Il existe de nombreuses façons de résoudre ce problème, mais elles sont un peu secondaires, comme le serait la refactorisation de notre code. Après tout, ce chapitre concerne le modèle Iterator, et non l'interface Iterable de Java. Mais la bonne nouvelle est que vous connaissez Iterable, vous connaissez sa relation avec l'interface Iterator de Java et le modèle Iterator. Alors, continuons, car nous avons une excellente implémentation même si nous ne tirons pas parti d'un peu de sucre syntaxique de la boucle for de Java.



Serious Coding

Vous avez probablement remarqué la méthode forEach() dans le menu Iterable. Elle est utilisée comme base pour la boucle for améliorée de Java, mais vous pouvez également l'utiliser directement avec Iterables. Voici comment cela fonctionne :

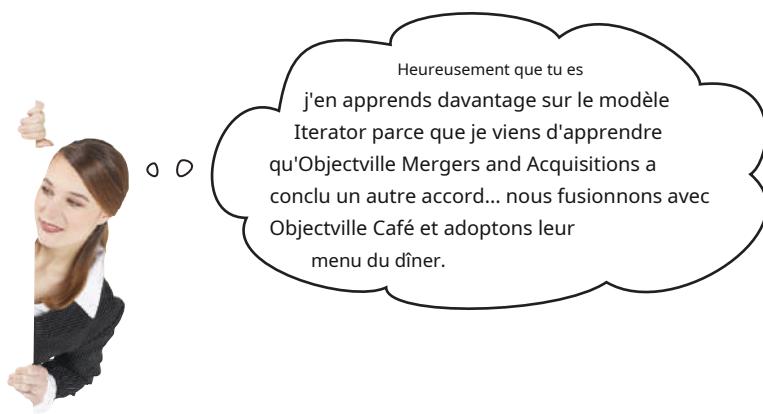
Voici un Iterable, dans ce cas notre Pancake House ArrayList d'éléments de menu.

Nous appelons forEach(...)

breakfastItems.forEach(element -> System.out.println(element));

... et en passant un lambda qui prend un menuItem et l'imprime simplement.

Ce code imprimera donc chaque élément de la collection.



Jetez un oeil au menu du café

Voici le menu du café. Il ne semble pas trop compliqué d'intégrer la classe CafeMenu dans notre framework... voyons cela.

```

classe publique CafeMenu {
    Carte<Chaîne, Élément de menu> menuItems = nouveau HashMap<Chaîne, Élément de menu>();

    public CafeMenu() {
        addItem("Burger végétarien et frites à l'air",
            « Burger végétarien sur un pain de blé entier, laitue, tomate et frites », vrai,
            3,99 $);
        addItem("Soupe du jour",
            "Une tasse de soupe du jour, accompagnée d'une salade",
            faux, 3,69);
        ajouter un élément("Burrito",
            "Un gros burrito, avec des haricots pinto entiers, de la salsa, du
            guacamole", vrai, 4,29);
    }

    public void addItem(Chaîne nom, Chaîne description,
        booléen végétarien, prix double)
    {
        MenuItem menuItem = new MenuItem(nom, description, végétarien, prix);
        menuItems.put(nom, menuItem);
    }
}

public Map<Chaîne, Élément de menu> getMenuItems() {
    retourner les éléments de menu;
}

```

CafeMenu n'implémente pas notre nouvelle interface de menu, mais cela est facilement résolu.

Le café stocke les éléments de son menu dans une HashMap. Est-ce que cela prend en charge Iterator ? Nous verrons bientôt...

Comme les autres menus, les éléments de menu sont initialisés dans le constructeur.

C'est ici que nous créons un nouveau MenuItem et l'ajoutons au HashMap menuItems.

La clé est le nom de l'article.

Nous n'en aurons plus besoin.



Sharpen your pencil

Avant de regarder la page suivante, notez rapidement les trois choses que nous devons faire avec ce code pour l'adapter à notre framework :

1.

2.

3.

Refonte du code du menu du café

Retravaillons le code de CafeMenu. Nous allons nous occuper de l'implémentation de l'interface Menu, et nous devons également nous occuper de la création d'un Iterator pour les valeurs stockées dans le HashMap. Les choses sont un peu différentes de ce que nous avons fait pour l'ArrayList ; regardez ça...

```
classe publique CafeMenu met en œuvre Menu {  
    Carte<Chaîne, Élément de menu> menuItems = nouveau HashMap<Chaîne, Élément de menu>();  
  
    public CafeMenu() {  
        // code constructeur ici  
    }  
  
    public void addItem(Chaîne nom, Chaîne description,  
                        booléen végétarien, prix double)  
    {  
        MenuItem menuItem = new MenuItem(nom, description, végétarien, prix);  
        menuItems.put(nom, menuItem);  
    }  
  
    public Map<Chaîne, Élément de menu> getMenuItems()  
    {  
        // retourner les éléments de menu;  
    }  
  
    public Iterator<MenuItem> createIterator()  
    {  
        renvoie menuItems.values().iterator();  
    }  
}
```

CafeMenu implémente l'interface Menu, donc la serveuse peut l'utiliser comme les deux autres menus.

Nous utilisons HashMap car c'est une structure de données courante pour stocker des valeurs.

Tout comme auparavant, nous pouvons nous débarrasser de getItems() afin de ne pas exposer l'implémentation de menuItems à la serveuse.

Et c'est ici que nous implementons la méthode createIterator(). Notez que nous n'obtenons pas d'itérateur pour l'ensemble du HashMap, mais uniquement pour les valeurs.



Le code de près

```
public Iterator<MenuItem> createIterator()  
{  
    renvoie menuItems.values().iterator();  
}
```

Nous obtenons d'abord les valeurs du HashMap, qui est simplement une collection de tous les objets du HashMap.

Heureusement, cette collection prend en charge la méthode iterator(), qui renvoie un objet de type java.util.Iterator.



Sommes-nous en train de violer le principe de Connaissances minimales ici ? Que peut-on qu'est-ce qu'on fait à ce sujet ?

Ajout du menu du café à la serveuse

Il est maintenant temps de modifier la serveuse pour qu'elle prenne en charge notre nouveau menu.

Maintenant que la serveuse attend des itérateurs, cela devrait être simple :

```
classe publique Serveuse {
    Menu crêpesMenuMaison;
    Menu dinerMenu;
    Menu caféMenu;
```

Le menu du café est transmis à la serveuse dans le constructeur avec les autres menus, et nous le stockons dans une variable d'instance.

```
Serveuse publique (Menu pancakeHouseMenu, Menu dinerMenu,
    ceci.pancakeHouseMenu = pancakeHouseMenu;
    ceci.dinerMenu = dinerMenu;
    ceci.caféMenu = caféMenu;
}
```

Menu caféMenu) {

```
public void printMenu() {
    Itérateur<MenuItem> pancakeIterator = pancakeHouseMenu.createIterator();
    Itérateur<MenuItem> dinerIterator = dinerMenu.createIterator();
    Itérateur<MenuItem> caféIterator = caféMenu.createIterator();

    System.out.println("MENU\n---\nPETIT DEJEUNER");
    printMenu(pancakeIterator);
    System.out.println("\nDÉJEUNER");
    printMenu(dinerIterator);
    System.out.println("\nDÎNER");
    printMenu(caféIterator);
}
```

Nous utilisons le menu du café pour notre menu du dîner. Pour l'imprimer, il suffit de créer l'itérateur et de le transmettre à printMenu(). C'est tout !

```
privé void printMenu(Iterator itérateur) {
    tandis que (itérateur.hasNext()) {
        MenuItem menuItem = itérateur.next();
        System.out.print(menuItem.getName() + ", ");
        System.out.print(menuItem.getPrice() + " -- ");
        System.out.println(menuItem.getDescription());
    }
}
```

Rien ne change ici.

Petit-déjeuner, déjeuner ET dîner

Mettions à jour notre essai routier pour nous assurer que tout fonctionne.

```
classe publique MenuTestDrive {  
    public static void main(String args[]) {  
        MenuCrêpesMaison pancakeHouseMenu = new MenuCrêpesMaison();  
        MenuDiner menuDinerMenu = new MenuDiner();  
        CaféMenu caféMenu = nouveau CaféMenu();  
        Crée un CaféMenu...  
        ... et passe-le à la serveuse.  
        Serveuse serveuse = nouvelle Serveuse(pancakeHouseMenu, dinerMenu,  
        (Menu du café) ;  
        ←  
        serveuse.printMenu();  
        Maintenant, lorsque nous imprimons, nous devrions voir les trois menus.  
    }  
}
```

Voici le test ; découvrez le nouveau menu du dîner du Café !

```
Aide sur la fenêtre d'édition de fichier  
% java DinerMenuTestDrive  
MENU  
----  
PETIT-DÉJEUNER  
Petit-déjeuner aux crêpes de K&B, 2,99 $ - Crêpes avec œufs brouillés et pain grillé  
Petit-déjeuner aux crêpes ordinaire, 2,99 $ - Crêpes avec œufs au plat et saucisses  
Crêpes aux myrtilles, 3,49 $ - Crêpes faites avec des myrtilles fraîches Gaufres, 3,59  
$ - Gaufres avec votre choix de myrtilles ou de fraises  
... et puis  
le restaurant  
menu...  
↓  
DÉJEUNER  
BLT végétarien, 2,99 $ - (Fakin') Bacon avec laitue et tomate sur du blé entier BLT, 2,99  
$ - Bacon avec laitue et tomate sur du blé entier  
Soupe du jour, 3.29 -- Soupe du jour, accompagnée d'une salade de pommes de terre  
Hot Dog, 3,05 $ - Un hot dog avec de la choucroute, de la relish, des oignons, garni de fromage.  
Légumes cuits à la vapeur et riz brun, 3,99 $ - Légumes cuits à la vapeur sur du riz brun. Pâtes,  
3,89 $ - Spaghetti à la sauce marinara et une tranche de pain au levain.  
... et enfin le  
nouveau café  
menu, tout avec  
le même  
code d'itération.  
DÎNER  
Soupe du jour, 3,69 $ - Une tasse de soupe du jour, accompagnée d'une salade Burrito, 4,29  
$ - Un grand burrito, avec des haricots pinto entiers, de la salsa, du guacamole Burger  
végétarien et frites à l'air, 3,99 $ - Burger végétarien sur un pain de blé entier,  
laitue, tomate et frites  
%
```

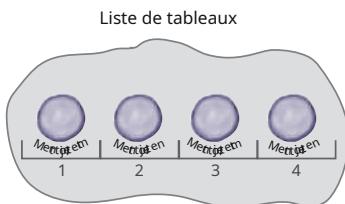
Qu'avons-nous fait ?



Nous voulions donner à la serveuse un moyen simple de parcourir les éléments du menu...

... et nous ne voulions pas qu'elle sache comment les éléments du menu sont mis en œuvre.

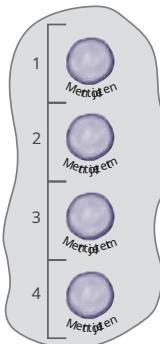
Nos éléments de menu avaient deux implémentations différentes et deux interfaces pour l'itération.



Tableau



Tableau

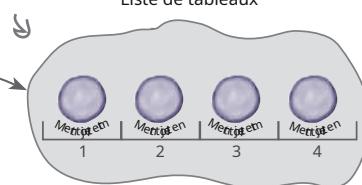


Nous avons découplé la Serveuse....

Nous avons donc donné à la serveuse un itérateur pour chaque type de groupe d'objets sur lequel elle devait effectuer une itération...

ArrayList a un itérateur intégré...

Liste de tableaux



... un pour
Liste de tableaux...



suivant()

j'stai du

... et un pour Array.

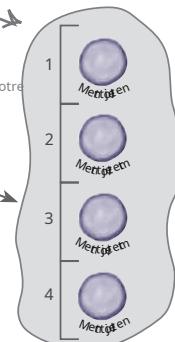
suivant()

j'stai du

Elle n'a plus à se soucier de l'implémentation que nous avons utilisée ; elle utilise toujours la même interface — Iterator — pour parcourir les éléments du menu. Elle a été découplée de l'implémentation.

... Tableau
n'a pas
un intégré
Itérateur donc
nous avons construit notre
propre.

Tableau

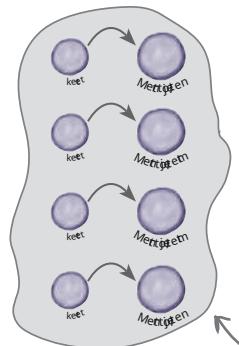


... et nous avons rendu la serveuse plus extensible



Nous avons facilement ajouté une autre implémentation du menu éléments, et puisque nous avons fourni un itérateur, la serveuse savait quoi faire.

HashMap



Créer un itérateur pour le HashMap
les valeurs étaient faciles;
quand tu appelles
`valeurs.iterator()`
vous obtenez un itérateur.

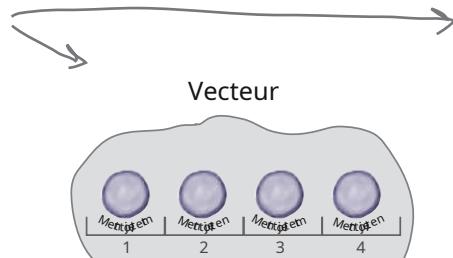
Mais il y a plus !

Java vous propose de nombreuses classes « Collection » qui vous permettent de stocker et de récupérer des groupes d'objets ; par exemple, Vector et

Liste chaînée.

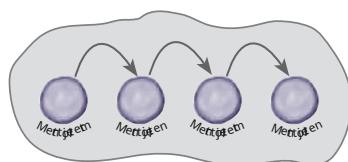
La plupart ont des différences interfaces.

Mais presque tous soutiennent une moyen d'obtenir un itérateur.



Et s'ils ne prennent pas en charge Iterator, ce n'est pas grave, car vous savez maintenant comment créer le vôtre.

Liste chaînée

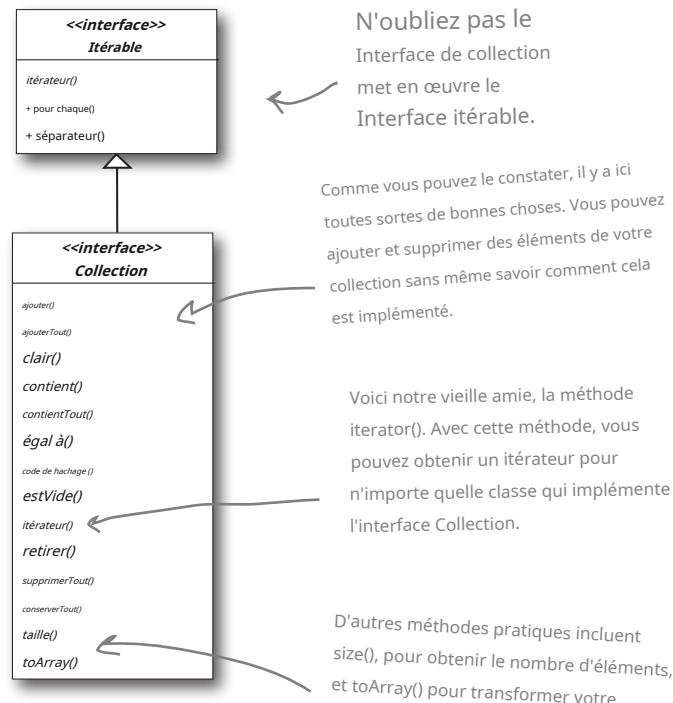


... et plus encore !

Itérateurs et collections

Nous avons utilisé quelques classes qui font partie du framework Java Collections. Ce « framework » n'est qu'un ensemble de classes et d'interfaces, dont ArrayList, que nous avons utilisé, et bien d'autres comme Vector, LinkedList, Stack et PriorityQueue. Chacune de ces classes implémente l'interface java.util.Collection, qui contient un ensemble de méthodes utiles pour manipuler des groupes d'objets.

Jetons un coup d'œil rapide à l'interface :



N'oubliez pas le Interface de collection met en œuvre le Interface itérable.

Comme vous pouvez le constater, il y a ici toutes sortes de bonnes choses. Vous pouvez ajouter et supprimer des éléments de votre collection sans même savoir comment cela est implémenté.

Voici notre vieille amie, la méthode `iterator()`. Avec cette méthode, vous pouvez obtenir un itérateur pour n'importe quelle classe qui implémente l'interface Collection.

D'autres méthodes pratiques incluent `size()`, pour obtenir le nombre d'éléments, et `toArray()` pour transformer votre collection en tableau.

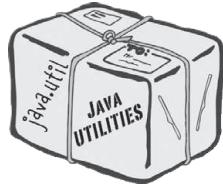
HashMap est l'une des rares classes qui indirectement prend en charge Iterator.

Watch it!

Comme vous l'avez vu quand nous *Après avoir implémenté le CafeMenu, vous pouvez en obtenir un itérateur, mais seulement en récupérant d'abord sa collection appelée valeurs. Si vous y réfléchissez, cela a du sens : le HashMap contient deux ensembles d'objets : les clés et les valeurs. Si nous voulons parcourir ses valeurs, nous devons d'abord les récupérer dans le HashMap, puis obtenir l'itérateur.*



L'avantage des collections et des itérateurs est que chaque objet Collection sait comment créer son propre itérateur. L'appel de `iterator()` sur une `ArrayList` renvoie un itérateur concret conçu pour les `ArrayLists`, mais vous n'avez jamais besoin de voir ou de vous soucier de la classe concrète qu'il utilise ; vous utilisez simplement l'interface `Iterator`.





Codes aimantés

Les chefs ont décidé qu'ils voulaient pouvoir alterner les éléments de leur menu du déjeuner ; en d'autres termes, ils proposeront certains éléments le lundi, le mercredi, le vendredi et le dimanche, et d'autres les mardi, jeudi et samedi. Quelqu'un a déjà écrit le code d'un nouvel itérateur de menu « Alternating » pour le dîner afin qu'il alterne les éléments du menu, mais elle l'a mélangé et l'a mis sur le réfrigérateur du restaurant pour plaisanter. Pouvez-vous le remettre en place ? Certaines accolades sont tombées par terre et elles étaient trop petites pour être ramassées, alors n'hésitez pas à en ajouter autant que vous le souhaitez.

```
MenuItem menuItem = éléments[position];
position = position + 2;
retourner menuItem;
```

```
importer java.util.Iterator;
importer java.util.Calendar;
```

```
Objet public suivant() {
```

```
{
```

```
public AlternatingDinerMenuItemIterator (éléments MenuItem [])
```

```
this.items = éléments;
position = Calendrier.JOUR_DE_LA_SEMAINE % 2;
```

```
implémente Iterator<MenuItem>
```

```
public void remove() {
```

```
MenuItem[] éléments;
position int;
```

```
}
```

```
classe publique AlternatingDinerMenuItemIterator
```

```
public boolen hasNext() {
```

```
lancer une nouvelle UnsupportedOperationException(
```

```
"L'itérateur de menu alternatif du restaurant ne prend pas en charge remove());
```

```
si (position >= éléments.length || éléments[position] == null) {
    retourner faux;
} autre {
    renvoie vrai ;
}
```

```
}
```



La serveuse est-elle prête pour le prime time ?

La serveuse a parcouru un long chemin, mais vous devez admettre que ces trois appels à `printMenu()` sont plutôt moches.

Soyons réalistes : chaque fois que nous ajoutons un nouveau menu, nous allons devoir ouvrir l'implémentation de `Waitress` et ajouter plus de code. Pouvez-vous dire « violer le principe d'ouverture et de fermeture » ?

Trois appels `createIterator()`.

```
public void printMenu() {
    Itérateur<MenuItem> pancakeIterator = pancakeHouseMenu.createIterator();
    Itérateur<MenuItem> dinerIterator = dinerMenu.createIterator(); Itérateur<MenuItem>
    cafeIterator = cafeMenu.createIterator();

    System.out.println("MENU\n----\nPETIT DEJEUNER");
    printMenu(pancakeIterator);

    System.out.println("\nDEJEUNER");
    printMenu(dinerIterator);

    System.out.println("\nDINER");
    printMenu(cafeIterator);
}
```

↑
←
↓
Trois appels à
imprimerMenu.

↑
Chaque fois que nous ajoutons ou supprimons un menu, nous
devrons ouvrir ce code pour apporter des modifications.

Ce n'est pas la faute de la serveuse. Nous avons fait un excellent travail en découpant l'implémentation du menu et en extrayant l'itération dans un itérateur. Mais nous gérons toujours les menus avec des objets séparés et indépendants. Nous avons besoin d'un moyen de les gérer ensemble.



La serveuse doit encore effectuer trois appels à `printMenu()`, un pour chaque menu. Pouvez-vous penser à un moyen de combiner les menus de manière à ce qu'un seul appel soit nécessaire ? Ou peut-être qu'un seul itérateur soit transmis à la serveuse pour parcourir tous les menus ?

Ce n'est pas si mal. Tout ce que nous devons faire est de regrouper les menus dans une ArrayList, puis de parcourir chaque menu. Le code de Waitress sera simple et gérera n'importe quel nombre de menus.



Il semble que le chef ait trouvé quelque chose. Essayons-le :

```
classe publique Serveuse {  
    Liste<Menu> menus ;  
  
    Serveuse publique(Liste<Menu> menus) {  
        ceci.menus = menus;  
    }  
  
    public void printMenu() {  
        Itérateur<Menu> menuIterator = menus.iterator();  
        while(menuIterator.hasNext()) {  
            Menu menu = menuIterator.next();  
            printMenu(menu.createIterator());  
        }  
    }  
  
    void printMenu(Iterator<MenuItem> itérateur) {  
        tandis que (itérateur.hasNext()) {  
            MenuItem menuItem = itérateur.next();  
            System.out.print(menuItem.getName() + ", ");  
            System.out.print(menuItem.getPrice() + " -- ");  
            System.out.println(menuItem.getDescription());  
        }  
    }  
}
```

Maintenant, nous prenons simplement une liste de menus, au lieu de chaque menu séparément.

Et nous parcourons les menus, en passant l'itérateur de chaque menu à la méthode printMenu() surchargée.

Pas de code changements ici.

Cela a l'air plutôt bien, même si nous avons perdu les noms des menus, mais nous pourrions ajouter les noms à chaque menu.

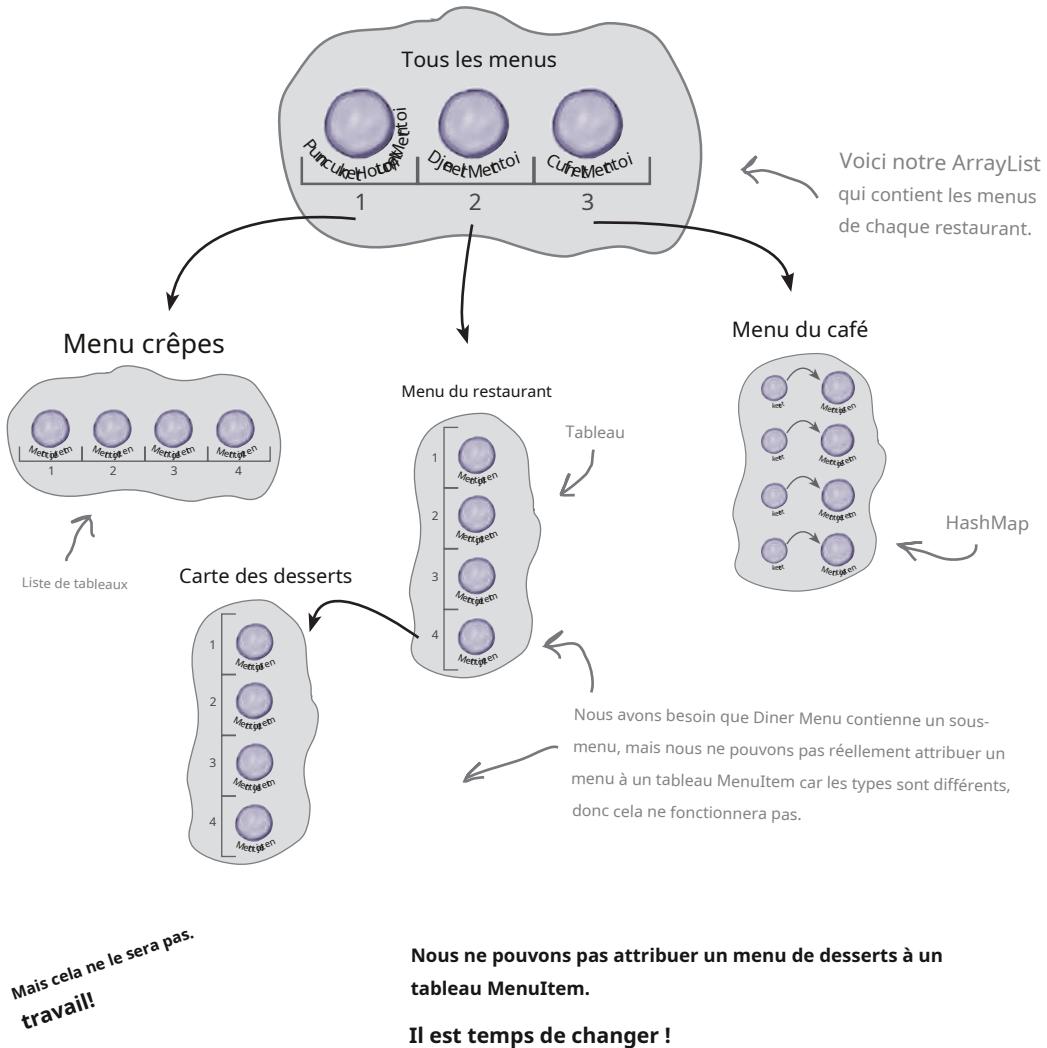
Juste au moment où nous pensions que c'était sûr...

Maintenant, ils veulent ajouter un sous-menu de desserts.

Ok, et maintenant ? Nous devons maintenant prendre en charge non seulement plusieurs menus, mais également des menus au sein des menus.

Ce serait bien si nous pouvions simplement faire du menu des desserts un élément de la collection DinerMenu, mais cela ne fonctionnera pas tel qu'il est actuellement implémenté.

Ce que nous voulons (quelque chose comme ça) :



De quoi avons-nous besoin ?

Le moment est venu de prendre une décision exécutive pour retravailler l'implémentation du chef en quelque chose de suffisamment général pour fonctionner sur tous les menus (et maintenant les sous-menus). C'est vrai, nous allons dire aux chefs que le moment est venu pour nous de réimplémenter leurs menus.

La réalité est que nous avons atteint un niveau de complexité tel que si nous ne retravaillons pas la conception maintenant, nous n'aurons jamais une conception capable d'accueillir d'autres acquisitions ou sous-menus.

Alors, de quoi avons-nous vraiment besoin dans notre nouvelle conception ?

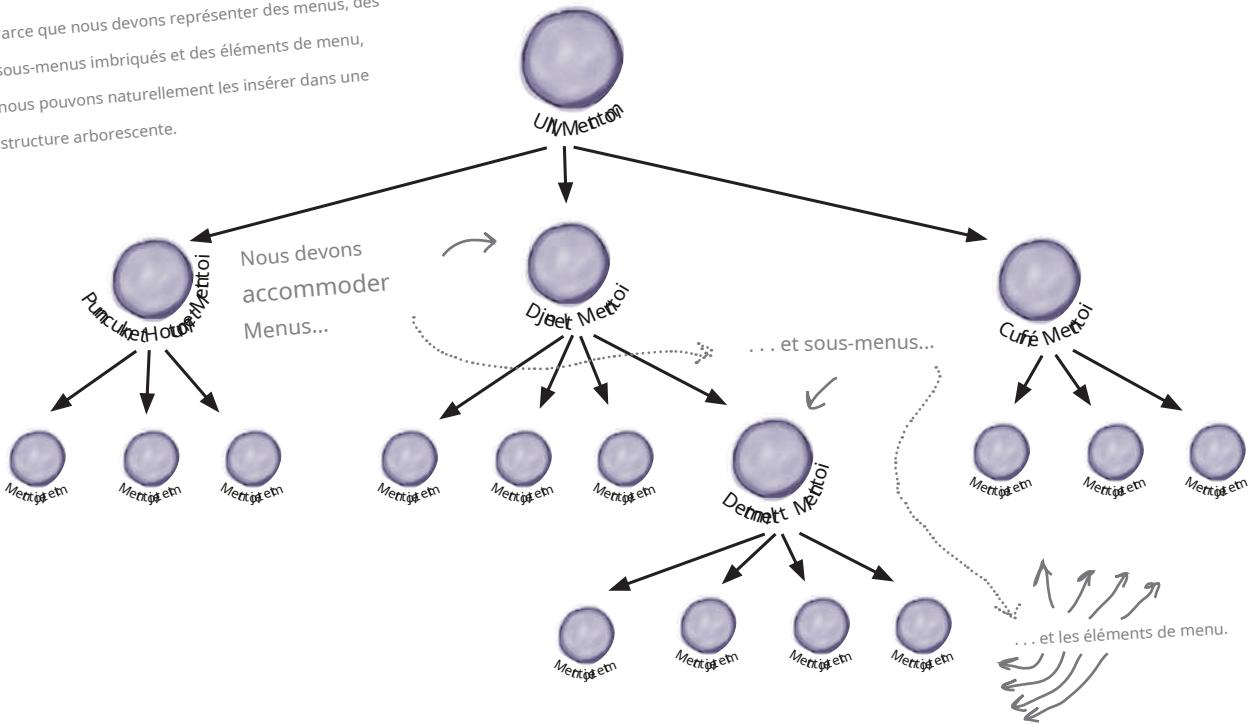
- Nous avons besoin d'une sorte de structure en forme d'arbre qui accueillera les menus, les sous-menus et les éléments de menu.
- Nous devons nous assurer de maintenir une manière de parcourir les éléments de chaque menu qui soit au moins aussi pratique que ce que nous faisons actuellement avec les itérateurs.
- Il se peut que nous ayons besoin de parcourir les éléments d'une manière plus flexible. Par exemple, nous pourrions avoir besoin d'itérer uniquement sur le menu des desserts du restaurant, ou nous pourrions avoir besoin d'itérer sur l'ensemble du menu du restaurant, y compris le sous-menu des desserts.

Il arrive un moment où nous devons refactoriser notre code pour qu'il puisse se développer. Ne pas le faire nous laisserait avec un code rigide et inflexible qui n'a aucune chance de germer un jour

nouvelle vie.

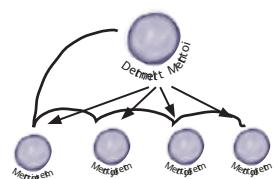
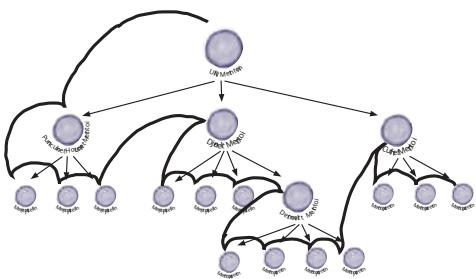


Parce que nous devons représenter des menus, des sous-menus imbriqués et des éléments de menu, nous pouvons naturellement les insérer dans une structure arborescente.



Nous devons encore pouvoir parcourir tous les éléments de l'arbre.

Nous devons également pouvoir naviguer de manière plus flexible, par exemple sur un seul menu.



Comment aborderiez-vous cette nouvelle étape dans nos exigences de conception ? Réfléchissez-y avant de tourner la page.

Le modèle composite défini

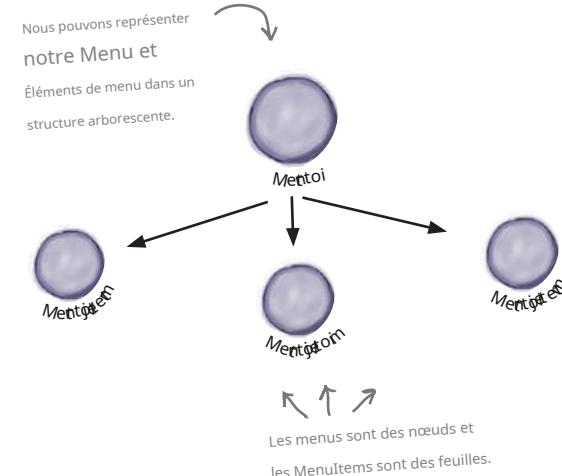
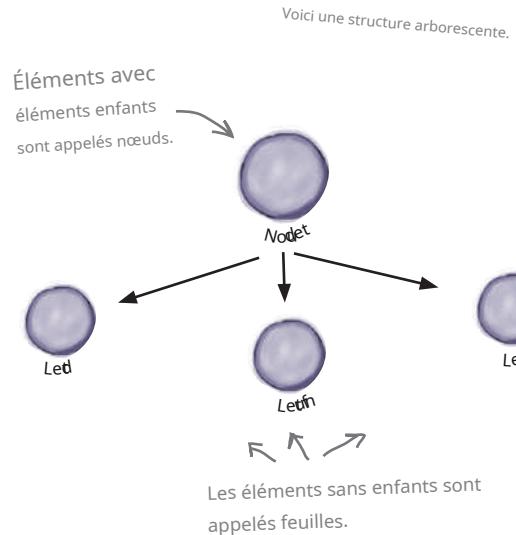
C'est vrai, nous allons introduire un autre modèle pour résoudre ce problème. Nous n'avons pas abandonné Iterator (il fera toujours partie de notre solution), mais le problème de la gestion des menus a pris une nouvelle dimension qu'Iterator ne résout pas. Nous allons donc prendre du recul et le résoudre avec le modèle composite.

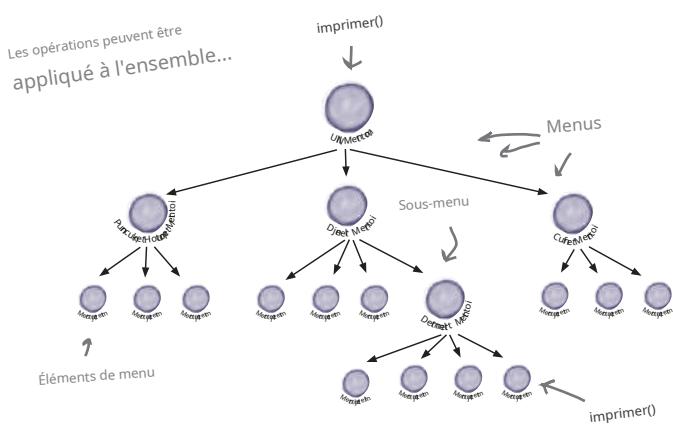
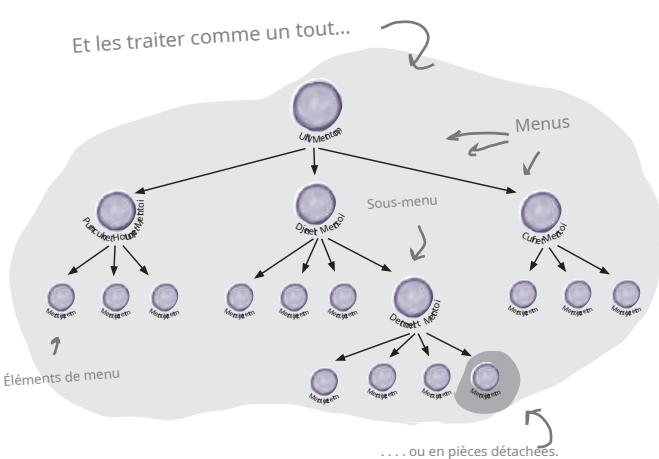
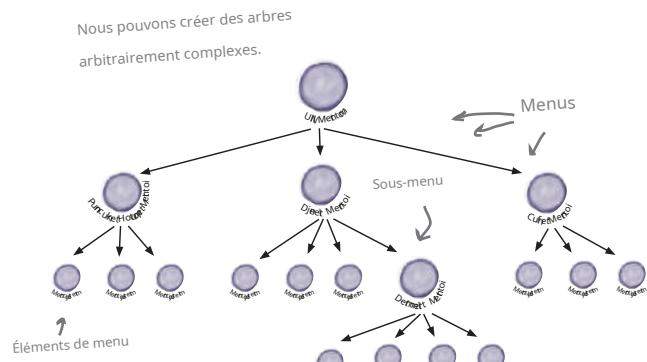
Nous n'allons pas tourner autour du pot sur ce modèle ; nous allons aller de l'avant et déployer la définition officielle maintenant :

Le modèle composite vous permet de composer des objets dans des structures arborescentes pour représenter des hiérarchies de parties-tout. Composite permet aux clients de traiter les objets individuels et les compositions d'objets de manière uniforme.

Pensons à cela en termes de nos menus : ce modèle nous donne un moyen de créer une structure arborescente capable de gérer un groupe imbriqué de menus et éléments de menu dans la même structure. En plaçant les menus et les éléments dans la même structure, nous créons une hiérarchie partie-tout, c'est-à-dire un arbre d'objets composé de parties (menus et éléments de menu) mais qui peut être traité comme un tout, comme un grand menu géant.

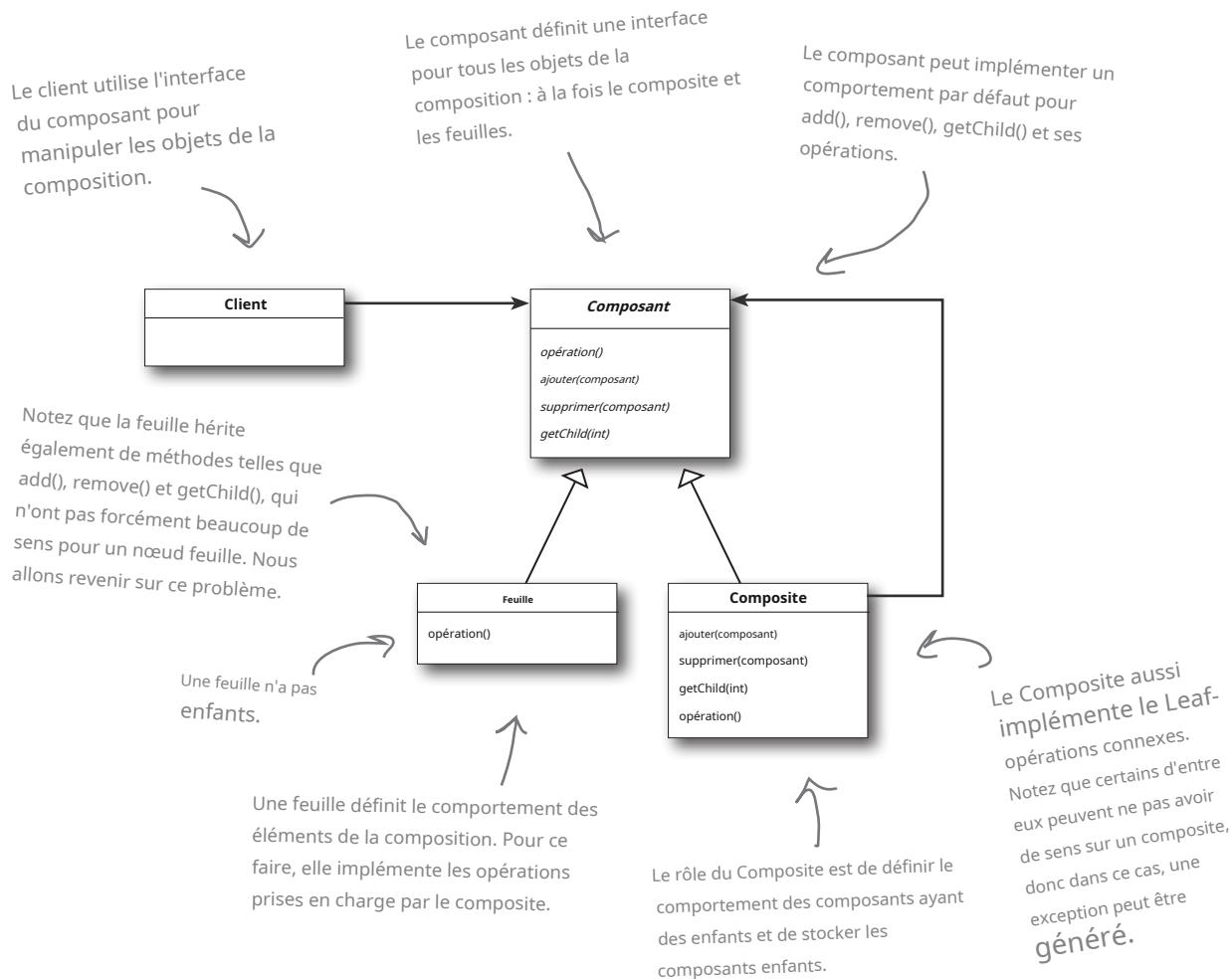
Une fois que nous avons notre menu über, nous pouvons utiliser ce modèle pour traiter « les objets individuels et les compositions de manière uniforme ». Qu'est-ce que cela signifie ? Cela signifie que si nous avons une structure arborescente de menus, de sous-menus et peut-être de sous-sous-menus ainsi que d'éléments de menu, alors tout menu est une « composition » car il peut contenir à la fois d'autres menus et d'autres éléments de menu. Les objets individuels ne sont que les éléments de menu, ils ne contiennent pas d'autres objets. Comme vous le verrez, l'utilisation d'une conception qui suit le modèle composite va nous permettre d'écrire un code simple qui peut appliquer la même opération (comme l'impression !) sur toute la structure du menu.





Le modèle composite nous permet de construire des structures d'objets sous forme d'arbres contenant à la fois des compositions d'objets et des objets individuels comme nœuds.

Utilisation d'un composite structure, nous pouvons appliquer les mêmes opérations sur les composites et objets individuels. Dans En d'autres termes, dans la plupart des cas, nous pouvons ignorer les différences entre compositions d'objets et objets individuels.



there are no Dumb Questions

Q: Composant, Composite, Arbres ? Je suis confus.

UN: Un composite contient des composants. Les composants sont de deux types : composites et éléments feuille. Cela vous semble récursif ? C'est le cas. Un composite contient un ensemble d'enfants ; ces enfants peuvent être d'autres composites ou éléments feuille.

Lorsque vous organisez les données de cette manière, vous obtenez une structure arborescente (en fait une structure arborescente à l'envers) avec un composite à la racine et des branches de composites poussant jusqu'aux feuilles.

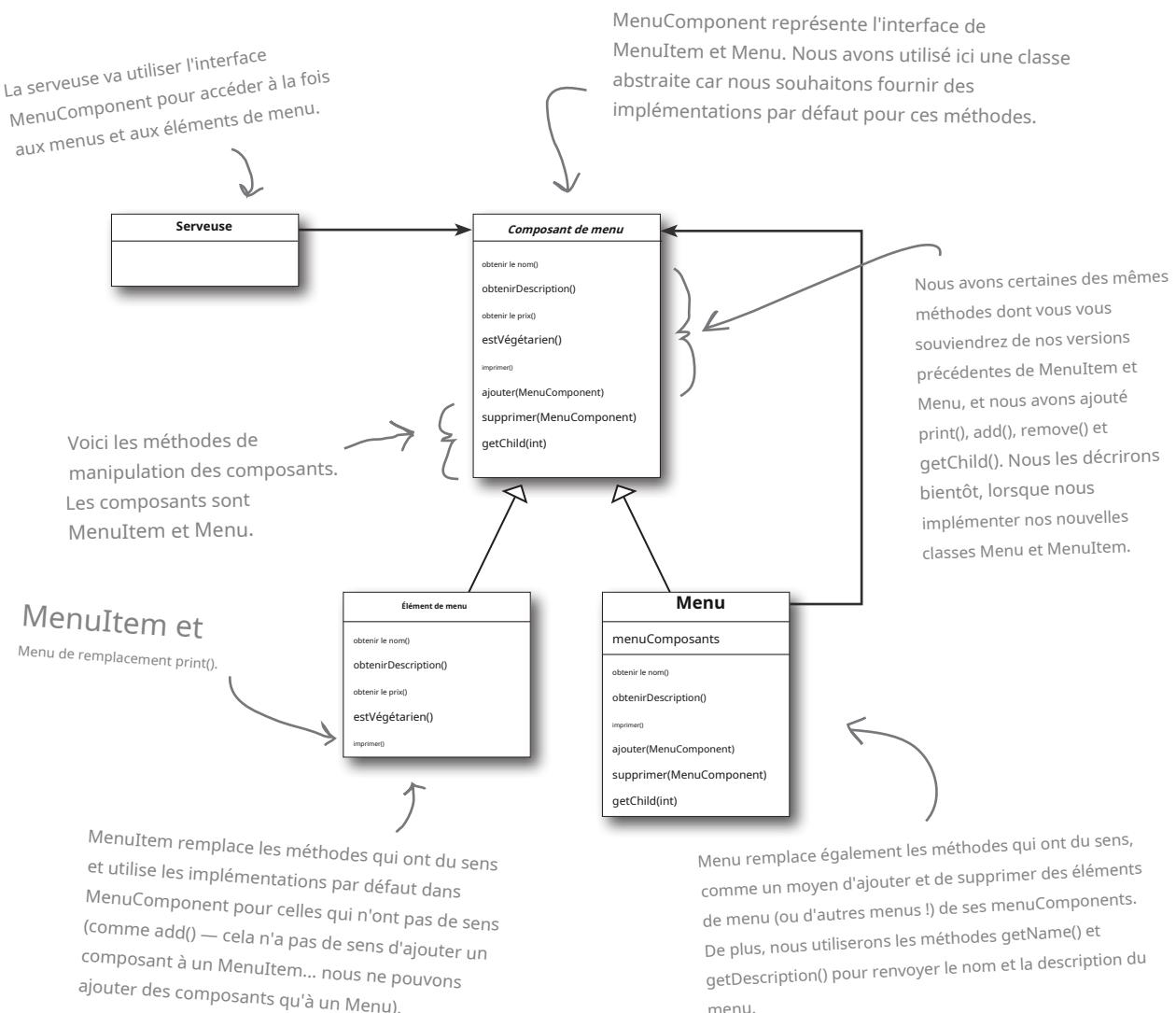
Q: Quel est le rapport avec les itérateurs ?

UN: N'oubliez pas que nous adoptons une nouvelle approche. Nous allons réimplémenter les menus avec une nouvelle solution : le modèle composite. Ne cherchez donc pas une transformation magique d'un itérateur en un composite. Cela dit, les deux fonctionnent très bien ensemble. Vous verrez bientôt que nous pouvons utiliser les itérateurs de plusieurs manières dans l'implémentation composite.

Concevoir des menus avec Composite

Alors, comment appliquer le modèle composite à nos menus ? Pour commencer, nous devons créer une interface de composant ; elle agit comme interface commune pour les menus et les éléments de menu et nous permet de les traiter de manière uniforme. En d'autres termes, nous pouvons appeler la même méthode sur les menus ou les éléments de menu.

Maintenant, cela ne se fera peut-être pas pour appeler certaines des méthodes d'un élément de menu ou d'un menu, mais nous pouvons nous en occuper, et nous le ferons dans un instant. Mais pour l'instant, examinons un schéma de la manière dont les menus vont s'intégrer dans une structure de modèle composite :



Implémentation de MenuComponent

Ok, nous allons commencer avec la classe abstraite MenuComponent ; rappelez-vous, le rôle du composant menu est de fournir une interface pour les feuilles et les nœuds composites. Maintenant, vous vous demandez peut-être : « MenuComponent ne joue-t-il pas deux rôles ? » C'est peut-être le cas et nous y reviendrons. Cependant, pour l'instant, nous allons fournir une implémentation par défaut des méthodes afin que si le MenuItem (la feuille) ou le Menu (le composite) ne souhaite pas implémenter certaines des méthodes (comme getChild() pour un nœud feuille), il puisse se rabattre sur un comportement de base :

```

Composant de menu
fournit la valeur par défaut
implémentations pour
chaque méthode.

classe abstraite publique MenuComponent {

    public void add(MenuComponent menuComponent) {
        lancer une nouvelle UnsupportedOperationException();
    }

    public void remove(MenuComponent menuComponent) {
        lancer une nouvelle UnsupportedOperationException();
    }

    public MenuComponent getChild(int i) {
        lancer une nouvelle UnsupportedOperationException();
    }

    chaîne publique getName() {
        lancer une nouvelle UnsupportedOperationException();
    }

    chaîne publique getDescription() {
        lancer une nouvelle UnsupportedOperationException();
    }

    public double getPrice() {
        lancer une nouvelle UnsupportedOperationException();
    }

    public boolean isVegetarian() {
        lancer une nouvelle UnsupportedOperationException();
    }

    public void print() {
        lancer une nouvelle UnsupportedOperationException();
    }
}

```

Tous les composants doivent implémenter l'interface MenuComponent ; cependant, comme les feuilles et les nœuds ont des rôles différents, nous ne pouvons pas toujours définir une implémentation par défaut pour chacun. **méthode qui a du sens. Parfois, le mieux que vous puissiez faire est de lancer une exception d'exécution.**

Étant donné que certaines de ces méthodes n'ont de sens que pour les MenuItems et que d'autres n'ont de sens que pour les Menus, l'implémentation par défaut est UnsupportedOperationException. De cette façon, si MenuItem ou Menu ne prend pas en charge une opération, il n'a rien à faire ; il peut simplement hériter de l'implémentation par défaut.

Nous avons regroupé les méthodes « composites », c'est-à-dire les méthodes permettant d'ajouter, de supprimer et d'obtenir des MenuComponents.

Voici les méthodes « opération » ; elles sont utilisées par les MenuItems. Il s'avère que nous pouvons également en utiliser quelques-unes dans Menu, comme vous le verrez dans quelques pages lorsque nous afficherons le code Menu.

print() est une méthode « opération » que nos menus et MenuItems sera implémenté, mais nous fournissons ici une opération par défaut.

Implémentation du MenuItem

Ok, essayons la classe MenuItem. N'oubliez pas qu'il s'agit de la classe feuille dans le diagramme composite et qu'elle implémente le comportement des éléments du composite.

```
classe publique MenuItem étend MenuComponent {
```

```
    Nom de la chaîne ;
```

```
    Description de la chaîne ;
```

```
    booléen végétarien;
```

```
    double prix;
```



Nous devons d'abord étendre le MenuComponent interface.

```
public MenuItem(Chaîne nom,
```

```
    Description de la chaîne,
```

```
    booléen végétarien,
```

```
    (prix double)
```



Le constructeur prend simplement le nom, la description, etc., et conserve une référence à tous ces éléments. Cela ressemble beaucoup à notre ancienne implémentation de MenuItem.

```
{
```

```
    ceci.nom = nom;
```

```
    this.description = description;
```

```
    this.vegetarian = végétarien; this.price
```

```
= prix;
```

```
}
```

```
chaîne publique getName() {
```

```
    renvoyer le nom;
```

```
}
```

```
chaîne publique getDescription() {
```

```
    description du retour;
```

```
}
```

```
public double getPrice() {
```

```
    prix de retour;
```

```
}
```

```
public boolean isVegetarian() {
```

```
    retour végétarien;
```

```
}
```

```
public void print() {
```

```
    System.out.print(" " + getName()); si
```

```
(isVegetarian()) {
```

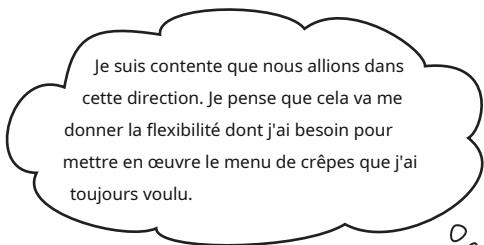
```
        Système.out.print("(v)");
```

```
}
```

```
    Système.out.println(", " + getPrice());
```

```
    Système.out.println(" - - " + getDescription());
```

```
}
```



Voici notre getter méthodes — tout comme notre implémentation précédente.

Ceci est différent de l'implémentation précédente. Ici, nous remplaçons la méthode print() dans la classe MenuComponent. Pour MenuItem, cette méthode imprime l'entrée de menu complète : nom, description, prix et si elle est végétarienne ou non.



Mise en œuvre du menu composite

Maintenant que nous avons le MenuItem, nous avons juste besoin de la classe composite, que nous appelons Menu. N'oubliez pas que la classe composite peut contenir des MenuItem ou autres menus. Il existe quelques méthodes de MenuComponent que cette classe n'implémente pas, getPrice() et isVegetarian(), car elles n'ont pas beaucoup de sens pour un menu.

```
Menu est également un MenuComponent,  
tout comme MenuItem.  
classe publique Menu étend MenuComponent {  
    Liste<MenuComponent> menuComponents = new ArrayList<MenuComponent>();  
    Chaîne nom;  
    Description de la chaîne ;  
  
    public Menu(Chaîne nom, Chaîne description) {  
        ceci.nom = nom;  
        ceci.description = description;  
    }  
  
    public void add(MenuComponent menuComponent) {  
        menuComponents.add(menuComponent);  
    }  
  
    public void remove(MenuComponent menuComponent) {  
        menuComponents.supprimer(menuComponent);  
    }  
  
    public MenuComponent getChild(int i) {  
        retourner menuComponents.get(i);  
    }  
  
chaîne publique getName() {  
    renvoyer le nom;  
    }  
  
chaîne publique getDescription() {  
    description du retour;  
    }  
  
public void print() {  
    System.out.print("\n" + getName());  
    System.out.println(", " + getDescription());  
    System.out.println("-----");  
    }
```

Le menu peut avoir n'importe quel nombre d'enfants de type MenuComponent. Nous utiliserons une ArrayList interne pour les contenir.

C'est différent de notre ancienne implémentation : nous allons donner à chaque menu un nom et une description. Avant, nous nous contentions d'avoir des classes différentes pour chaque menu.

Voici comment ajouter des MenuItem ou d'autres menus à un menu. Étant donné que les MenuItem et les Menus sont des MenuComponents, nous n'avons besoin que d'une seule méthode pour faire les deux.

Vous pouvez également supprimer un MenuComponent ou obtenir un MenuComponent.

Voici les méthodes getter pour obtenir le nom et la description.

Notez que nous ne remplaçons pas getPrice() ou isVegetarian() car ces méthodes n'ont pas de sens pour un menu (bien que vous puissiez argumenter que isVegetarian() pourrait avoir du sens). Si quelqu'un essaie d'appeler ces méthodes sur un menu, il obtiendra une UnsupportedOperationException.

Pour imprimer le Menu, nous imprimons son nom et sa description.



Attends une seconde, je ne sais pas.
comprendre l'implémentation de print(). Je pensais que j'étais censé pouvoir appliquer les mêmes opérations à un composite qu'à une feuille. Si j'applique print() à un composite avec cette implémentation, tout ce que j'obtiens est un simple nom de menu et une description. Je ne obtenir une impression du COMPOSITE.

Bonne prise. Étant donné que Menu est un composite et contient à la fois des MenuItem et d'autres Menus, sa méthode print() doit imprimer tout ce qu'il contient. Si ce n'est pas le cas, nous devrons parcourir l'intégralité du composite et imprimer chaque élément nous-mêmes. Cela va à l'encontre de l'objectif d'une structure composite.

Comme vous allez le voir, implémenter correctement print() est facile car nous pouvons compter sur chaque composant pour pouvoir s'imprimer lui-même. Tout est merveilleusement récursif et génial. Regardez ça :

Correction de la méthode print()

```
classe publique Menu étend MenuComponent {
```

```
    Liste<MenuComponent> menuComponents = new ArrayList<MenuComponent>();
```

```
    Chaîne nom;
```

```
    Description de la chaîne ;
```

```
// code constructeur ici
```

```
// autres méthodes ici
```

```
public void print() {
```

```
    System.out.print("\n" + getName());
```

```
    System.out.println(", " + getDescription());
```

```
    System.out.println("-----");
```

```
    pour (MenuComponent menuComponent : menuComponents) {
        menuComponent.print();
    }
```

```
}
```

Tout ce que nous devons faire est de modifier la méthode print() pour qu'elle imprime non seulement les informations sur ce menu, mais tous les composants de ce menu : les autres menus et les éléments de menu.

Regardez ! Nous pouvons utiliser un itérateur dans les coulisses de la boucle for améliorée. Nous l'utilisons pour parcourir tous les composants du menu... il peut s'agir d'autres menus ou d'éléments de menu.

Étant donné que les menus et les éléments de menu implémentent tous deux print(), nous appelons simplement print() et le reste leur appartient.

REMARQUE : si, au cours de cette itération, nous rencontrons un autre objet Menu, sa méthode print() démarra une autre itération, et ainsi de suite.

Préparation pour un essai routier...

Il est temps de tester ce code, mais nous devons d'abord mettre à jour le code de la serveuse, après tout, elle est le client principal de ce code :

classe publique Serveuse {

Composant de menu tous les menus ;



Oui ! Le code de la serveuse est vraiment aussi simple que ça. Il ne nous reste plus qu'à lui transmettre le composant de menu de niveau supérieur, celui qui contient tous les autres menus. Nous l'avons appelé allMenus.

Serveuse publique (MenuComponent allMenus) {

ceci.allMenus = tous les menus;

}



Pour imprimer toute la hiérarchie du menu (tous les menus et tous les éléments du menu), il lui suffit d'appeler print() sur le menu de niveau supérieur.

public void printMenu() {

tous les menus.print();

}

}

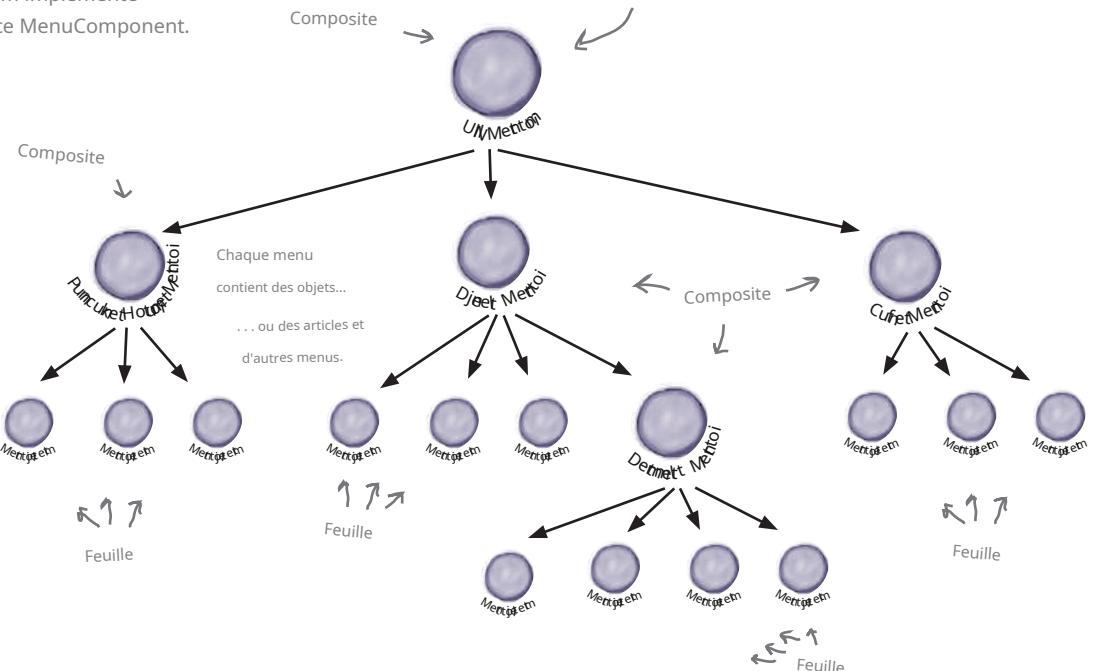
Nous allons avoir une serveuse heureuse.

Bon, une dernière chose avant d'écrire notre test. Voyons à quoi ressemblera le menu composite au moment de l'exécution :

Chaque menu et MenuItem implémente l'interface MenuComponent.

Composite

Le menu de niveau supérieur contient tous les menus et éléments.



Passons maintenant à l'essai routier...

Bon, il ne nous reste plus qu'à faire un essai. Contrairement à notre version précédente, nous allons gérer toute la création du menu lors de l'essai. Nous pourrions demander à chaque chef de nous donner son nouveau menu, mais testons d'abord tout. Voici le code :

```

classe publique MenuTestDrive {
    public static void main(String args[]){
        Composant de menu pancakeHouseMenu =
            nouveau Menu("MENU PANCAKE HOUSE", "Petit déjeuner");
        MenuComponent dinerMenu =
            nouveau Menu("MENU DU DINER",
            "Déjeuner"); MenuComponent cafeMenu =
            nouveau Menu("MENU CAFE", "Dîner");
        MenuComponent dessertMenu =
            nouveau Menu("MENU DESSERT", "Dessert bien sûr !");
        MenuComponent allMenus = new Menu("TOUS LES MENUS", "Tous les menus combinés");

        allMenus.add(pancakeHouseMenu); ← Commençons d'abord par créer tous les objets du menu.
        allMenus.add(dinerMenu); ← Nous avons également besoin d'un menu de niveau supérieur que nous nommerons allMenus.
        tous les menus.add(cafeMenu); ← Nous utilisons la méthode Composite add() pour ajouter chaque menu au menu de niveau supérieur, allMenus.

        // ajouter des éléments de menu ici ← Il nous faut maintenant ajouter tous les éléments du menu. Voici un exemple ; pour le reste, consultez le code source complet.

        dinerMenu.add(nouveau MenuItem(
            "Pâtes",
            "Spaghetti à la sauce marinara et une tranche de pain au levain", c'est vrai,
            3.89)); ← Et nous ajoutons également un menu à un menu. Tout ce qui importe à dinerMenu, c'est que tout ce qu'il contient, qu'il s'agisse d'un élément de menu ou d'un menu, soit un MenuComponent.

        dinerMenu.add(dessertMenu); ← Ajoutez un peu de tarte aux pommes au menu des desserts...
        dessertMenu.add(nouveau MenuItem(
            "Tarte aux pommes",
            "Tarte aux pommes à la croûte feuilletée, garnie de glace à la vanille", c'est vrai,
            1.59)); ← Une fois que nous avons construit toute notre hiérarchie de menu, nous remettons le tout à la serveuse, et comme vous l'avez vu, c'est aussi simple que de faire une tarte aux pommes pour elle de l'imprimer.

        // ajouter plus d'éléments de menu ici

        Serveuse serveuse = new Waitress(allMenus);
        serveuse.printMenu(); ←
    }
}

```

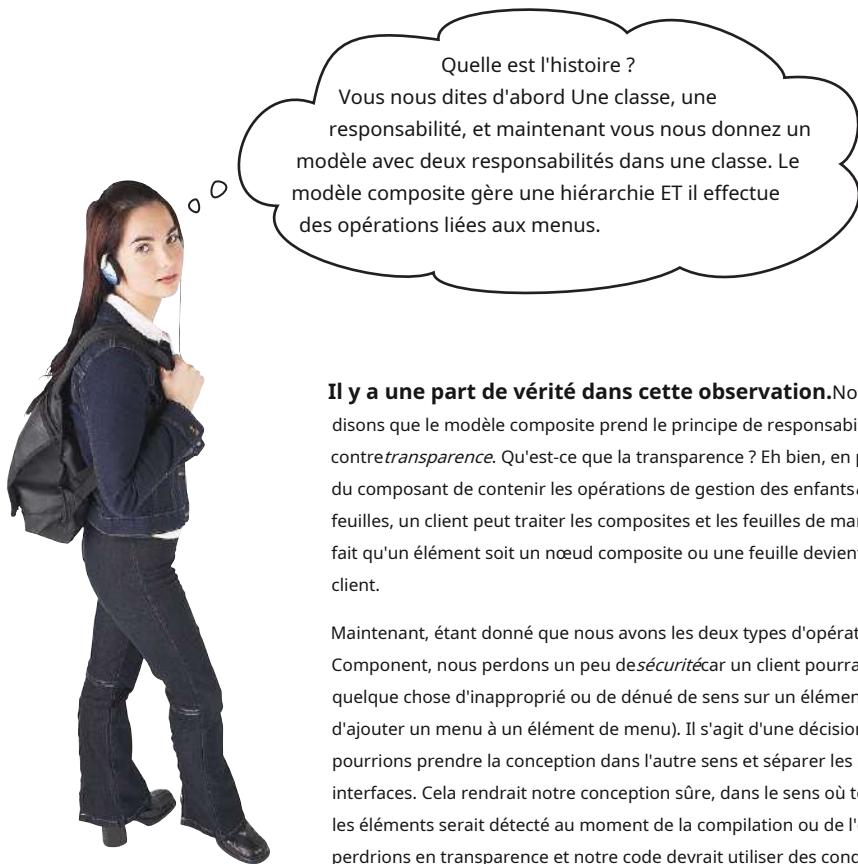
Préparation pour un essai routier...

REMARQUE : cette sortie est basée sur la source complète.

```
Aide sur la fenêtre d'édition de fichier  
% java MenuTestDrive  
TOUS LES MENUS, Tous les menus confondus  
-----  
MENU PANCAKE HOUSE, Petit déjeuner  
-----  
Petit déjeuner aux crêpes de K&B(v), 2,99  
-- Crêpes aux œufs brouillés et pain grillé  
Petit déjeuner régulier aux crêpes, 2,99 $  
-- Crêpes aux œufs au plat, saucisse  
Crêpes aux myrtilles(v), 3,49  
-- Crêpes à base de myrtilles fraîches et de sirop de myrtilles  
Gaufres(v), 3,59  
-- Gaufres avec votre choix de myrtilles ou de fraises  
MENU DINER, Déjeuner  
-----  
BLT(v) végétarien, 2,99  
-- (Fakin') Bacon avec laitue et tomate sur du blé entier  
BLT, 2,99  
- Bacon avec laitue et tomate sur pain de blé entier  
Soupe du jour, 3,29  
-- Un bol de soupe du jour, accompagné d'une salade de pommes de terre  
Hot-dog, 3,05  
-- Un hot-dog, avec de la choucroute, de la relish, des oignons, garni de fromage  
Légumes cuits à la vapeur et riz brun (v), 3,99  
-- Légumes cuits à la vapeur sur riz brun  
Pâtes(v), 3,89  
-- Spaghetti à la sauce marinara et une tranche de pain au levain  
CARTE DESSERT, Dessert bien sûr !  
-----  
Tarte  
aux pommes (v), 1,59  
-- Tarte aux pommes avec une croûte feuilletée, garnie de glace à la vanille  
Gâteau au fromage (v), 1,99  
-- Cheesecake crémeux de New York, avec une croûte au chocolat Graham  
Sorbet(v), 1,89  
-- Une boule de framboise et une boule de citron vert  
MENU CAFÉ, Dîner  
-----  
Burger végétarien et frites à l'air(v), 3,99  
- Burger végétarien sur pain de blé entier, laitue, tomate et frites  
Soupe du jour, 3,69  
-- Une tasse de soupe du jour, accompagnée d'une salade  
Burrito(v), 4,29  
-- Un grand burrito, avec des haricots pinto entiers, de la salsa et du guacamole  
%
```

Voici tous nos menus... nous avons imprimé tout cela simplement en appelant print() sur le menu de niveau supérieur.

Le nouveau carte des desserts est imprimé quand nous sommes imprimer tous les Menu du dîner composants.



Quelle est l'histoire ?
Vous nous dites d'abord Une classe, une responsabilité, et maintenant vous nous donnez un modèle avec deux responsabilités dans une classe. Le modèle composite gère une hiérarchie ET il effectue des opérations liées aux menus.

Il y a une part de vérité dans cette observation. Nous pourrions disons que le modèle composite prend le principe de responsabilité unique et l'échange contretransparence. Qu'est-ce que la transparence ? Eh bien, en permettant à l'interface du composant de contenir les opérations de gestion des enfants et des opérations sur les feuilles, un client peut traiter les composites et les feuilles de manière uniforme ; ainsi, le fait qu'un élément soit un nœud composite ou une feuille devient transparent pour le client.

Maintenant, étant donné que nous avons les deux types d'opérations dans la classe Component, nous perdons un peu de sécurité car un client pourrait essayer de faire quelque chose d'inapproprié ou de dénué de sens sur un élément (comme essayer d'ajouter un menu à un élément de menu). Il s'agit d'une décision de conception ; nous pourrions prendre la conception dans l'autre sens et séparer les responsabilités en interfaces. Cela rendrait notre conception sûre, dans le sens où tout appel inapproprié sur les éléments serait détecté au moment de la compilation ou de l'exécution, mais nous perdrons en transparence et notre code devrait utiliser des conditions et l'instance de opérateur.

Donc, pour revenir à votre question, il s'agit d'un cas classique de compromis. Nous sommes guidés par des principes de conception, mais nous devons toujours observer l'effet qu'ils ont sur nos conceptions. Parfois, nous faisons délibérément des choses d'une manière qui semble enfreindre le principe. Dans certains cas, cependant, c'est une question de perspective ; par exemple, il peut sembler incorrect d'avoir des opérations de gestion des enfants dans les feuilles (comme add(), remove() et getChild()), mais là encore, vous pouvez toujours changer de perspective et voir une feuille comme un nœud avec zéro enfant.



Les modèles dévoilés

Ceinterview de la semaine :

Le modèle composite, sur les problèmes de mise en œuvre

Tête la première : Nous sommes ici ce soir pour parler avec le Composite Pattern. Pourquoi ne nous parlez-vous pas un peu de vous, Composite ?

Composite: Bien sûr... Je suis le modèle à utiliser lorsque vous avez des collections d'objets avec des relations tout-partie et que vous souhaitez pouvoir traiter ces objets de manière uniforme.

Tête la première : Ok, plongeons directement ici... qu'entendez-vous par relations tout-partie ?

Composite: Imaginez une interface utilisateur graphique (GUI) ; vous y trouverez souvent un composant de niveau supérieur comme un cadre ou un panneau, contenant d'autres composants, comme des menus, des volets de texte, des barres de défilement et des boutons. Votre interface graphique se compose donc de plusieurs parties, mais lorsque vous l'affichez, vous la considérez généralement comme un tout. Vous dites au composant de niveau supérieur de s'afficher et vous comptez sur ce composant pour afficher toutes ses parties. Nous appelons les composants qui contiennent d'autres composants, *objets composites*, et les composants qui ne contiennent pas d'autres composants *objets en forme de feuille*.

Tête la première : Est-ce ce que vous voulez dire par traiter les objets de manière uniforme ? En ayant des méthodes communes, vous pouvez faire appel à des composites et à des feuilles ?

Composite: C'est vrai. Je peux dire à un objet composite de s'afficher ou à un objet feuille de s'afficher et il fera ce qu'il faut. L'objet composite s'affichera en disant à tous ses composants de s'afficher.

Tête la première : Cela implique que chaque objet possède la même interface. Que se passe-t-il si vous avez des objets dans votre composite qui font des choses différentes ?

Composite: Pour que le composite fonctionne de manière transparente pour le client, vous devez implémenter la même interface pour tous les objets du composite. Sinon, le client doit se soucier de l'interface implémentée par chaque objet, ce qui va à l'encontre de l'objectif. Évidemment, cela signifie que vous aurez parfois des objets pour lesquels certains appels de méthode n'ont pas de sens.

Tête la première : Alors, comment gérez-vous cela ?

Composite: Il existe plusieurs façons de gérer cela. Parfois, vous pouvez simplement ne rien faire ou renvoyer null ou false, selon ce qui convient à votre application. D'autres fois, vous souhaiterez être plus proactif et lancer une exception. Bien entendu, le client doit alors être prêt à faire un peu de travail et à s'assurer que l'appel de méthode n'a pas fait quelque chose d'inattendu.

Tête la première : Mais si le client ne sait pas à quel type d'objet il a affaire, comment pourra-t-il savoir quels appels effectuer sans vérifier le type ?

Composite: Si vous êtes un peu créatif, vous pouvez structurer vos méthodes de manière à ce que les implémentations par défaut fassent quelque chose qui a du sens. Par exemple, si le client appelle getChild() sur le composite, cela a du sens. Et cela a également du sens sur une feuille, si vous considérez la feuille comme un objet sans enfant.

Tête la première : Ah... intelligent. Mais j'ai entendu dire que certains clients sont tellement préoccupés par ce problème qu'ils exigent des interfaces distinctes pour différents objets afin de ne pas être autorisés à effectuer des appels de méthodes absurdes. Est-ce toujours le modèle composite ?

Composite: Oui. C'est une version beaucoup plus sûre du modèle composite, mais elle nécessite que le client vérifie le type de chaque objet avant de passer un appel afin que l'objet puisse être correctement lancé.

Tête la première : Parlez-nous un peu plus de la façon dont ces objets composites et feuilles sont structurés.

Composite: Il s'agit généralement d'une structure arborescente, d'une sorte de hiérarchie. La racine est le composite de niveau supérieur et tous ses enfants sont soit des composites, soit des feuilles.

Tête la première : Les enfants pointent-ils parfois du doigt leurs parents ?

Composite: Oui, un composant peut avoir un pointeur vers un parent pour faciliter la traversée de la structure. Et, si

vous avez une référence à un enfant et vous devez le supprimer, vous devrez demander au parent de supprimer l'enfant. Le fait d'avoir la référence au parent facilite également cette tâche.

Tête la première : Il y a vraiment beaucoup de choses à prendre en compte dans votre implémentation. Y a-t-il d'autres problèmes auxquels nous devrions penser lors de la mise en œuvre du modèle composite ?

Composite: En fait, il y en a. L'une d'entre elles est l'ordre des enfants. Que se passe-t-il si vous avez un composite qui doit conserver ses enfants dans un ordre particulier ? Vous aurez alors besoin d'un schéma de gestion plus sophistiqué pour ajouter et supprimer des enfants, et vous devrez faire attention à la façon dont vous parcourez la hiérarchie.

Tête la première : Un bon point auquel je n'avais pas pensé.

Composite: Et avez-vous pensé à la mise en cache ?

Tête la première : Mise en cache ?

Composite: Oui, la mise en cache. Parfois, si la structure composite est complexe ou coûteuse à parcourir, il est utile d'implémenter la mise en cache des noeuds composites. Par exemple, si vous parcourez constamment un composite et tous ses enfants pour calculer un résultat, vous pouvez implémenter un cache qui stocke le résultat temporairement pour économiser les traversées.

Tête la première : Eh bien, les modèles composites sont bien plus complexes que je ne l'aurais jamais imaginé. Avant de conclure, une dernière question : quelle est, selon vous, votre plus grande force ?

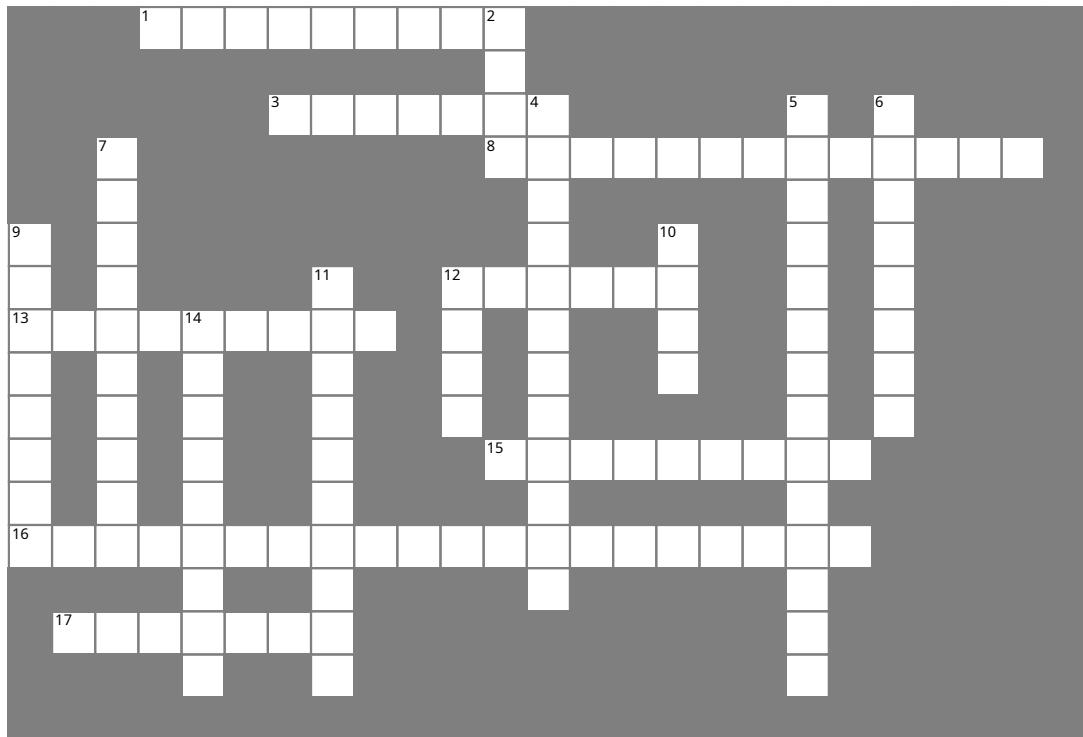
Composite: Je pense que je dirais sans hésiter que c'est la simplification de la vie de mes clients. Mes clients n'ont pas à se soucier de savoir s'ils ont affaire à un objet composite ou à un objet feuille, ils n'ont donc pas besoin d'écrire des instructions if partout pour s'assurer qu'ils appellent les bonnes méthodes sur les bons objets. Souvent, ils peuvent effectuer un appel de méthode et exécuter une opération sur une structure entière.

Tête la première : Cela semble être un avantage important. Il ne fait aucun doute que vous êtes un modèle utile à avoir sous la main pour collecter et gérer des objets. Et, avec cela, nous n'avons plus de temps. Merci beaucoup de nous avoir rejoint et revenez bientôt pour un autre Patterns Exposed.



Mots croisés sur les modèles de conception

Réfléchissez à ces mots croisés composites.



À TRAVERS

1. Collection et Iterator sont dans ce package.
3. Cette classe prend en charge indirectement Iterator.
8. Les itérateurs sont généralement créés à l'aide de ce modèle (deux mots).
12. Une classe ne devrait avoir qu'une seule raison de faire cela.
13. Nous avons encapsulé ceci.
15. Les packages d'interface utilisateur utilisent souvent ce modèle pour leurs composants.
16. Nom du principe qui énonce une seule responsabilité par classe (deux mots).
17. Ce menu nous a amené à modifier toute notre implémentation.

VERS LE BAS

2. N'a pas d'enfants.
4. Fusionné avec le Diner (deux mots).
5. Le modèle Iterator dissocie le client du _____ de l'agrégat.
6. Un objet distinct qui peut parcourir une collection.
7. Les valeurs HashMap et ArrayList implémentent toutes deux cette interface.
9. Nous l'avons équipée de Java.
10. Un composant peut être un composite ou ceci.
11. Un composite les contient.
12. Troisième société acquise.
14. Nous avons supprimé le PancakeHouseMenuIterator car cette classe fournit déjà un itérateur.



Associez chaque motif à sa description :

Modèle	Description
Stratégie	Les clients traitent les collections d'objets et les objets individuels de manière uniforme
Adaptateur	Fournit un moyen de parcourir une collection d'objets sans exposer le implémentation de la collection
Itérateur	Simplifie l'interface d'un groupe de classes
Façade	Modifie l'interface d'une ou plusieurs classes
Composite	Permet à un groupe d'objets d'être notifié lorsque certains états changent
Observateur	Encapsule les comportements interchangeables et utilise la délégation pour décider lequel utiliser



Des outils pour votre boîte à outils de conception

Deux nouveaux modèles pour votre boîte à outils : deux excellentes façons de gérer des collections d'objets.

Principes OO

- Encapsuler ce qui varie
- Privilégiez la composition plutôt que l'héritage.
- Programmez vers des interfaces, pas vers des implémentations.
- Efforcez-vous d'obtenir des conceptions faiblement couplées entre les objets qui interagissent.
- Les cours devraient être ouverts pour prolongation mais fermés pour modification.
- Dépendez des abstractions. Ne dépendez pas de classes concrètes.
- Parlez uniquement à vos amis.
- Ne nous appelez pas, nous vous appellons.
- Une classe ne devrait avoir qu'une seule raison de changer.

Notions de base sur OO

- Abstraction
- Encapsulation
- Polymorphisme
- Héritage

Encore un autre point important principe basé sur le changement dans une conception.

Modèles OO

- Strategy
- Factory
- Adapter
- Decorator
- Composite
- Singleton

Encore un deux pour un chapitre.

mat

Ce modèle permet de décliner plusieurs types de structures et de leur donner des propriétés partagées. C'est une composition d'objets dans lesquels certains sont des parties d'un tout. Il permet de traiter des objets individuels et compositions d'objets de manière uniforme.



BULLET POINTS

- f Un itérateur permet d'accéder aux éléments d'un agrégat sans exposer sa structure interne.
 - f Un itérateur prend en charge la tâche d'itérer sur un agrégat et l'encapsule dans un autre objet.
 - f Lorsque nous utilisons un itérateur, nous déchargeons l'agrégat de la responsabilité de prendre en charge les opérations de parcours de ses données.
 - f Un itérateur fournit une interface commune pour parcourir les éléments d'un agrégat, vous permettant d'utiliser le polymorphisme lors de l'écriture de code qui utilise les éléments de l'agrégat.
 - f L'interface Iterable fournit un moyen d'obtenir un itérateur et active la boucle for améliorée de Java.
 - f Nous devrions nous efforcer d'assigner une seule responsabilité à chaque classe.
 - f Le modèle composite permet aux clients de traiter les composites et les objets individuels de manière uniforme. erajon, Un composant est un objet quelconque dans une structure composite. Les composants peuvent être d'autres composites ou des feuilles.
 - f La mise en œuvre de Composite implique de nombreux compromis de conception. Vous devez trouver un équilibre entre transparence et sécurité en fonction de vos besoins.



Sharpen your pencil

Solution

Sur la base de notre implémentation de printMenu(), lesquelles des affirmations suivantes s'appliquent ?

- A. Nous codons pour les implémentations concrètes de PancakeHouseMenu et DinerMenu, pas pour une interface.
- B. La serveuse n'implémente pas l'API Java Waitress et elle n'est donc pas adhérent à une norme.
- C. Si nous décidions de passer de l'utilisation de DinerMenu à un autre type de menu qui implémentait sa liste d'éléments de menu avec une table de hachage, nous devrions modifier beaucoup de code dans Waitress.
- D. La serveuse doit savoir comment chaque menu représente sa collection interne d'éléments de menu ; cela viole l'encapsulation.
- E. Nous avons un code en double : la méthode printMenu() nécessite deux boucles distinctes pour itérer sur les deux types de menus différents. Et si nous ajoutons un troisième menu, nous aurions encore une autre boucle.
- F. L'implémentation n'est pas basée sur MXML (Menu XML) et n'est donc pas aussi interopérable qu'elle devrait l'être.



Sharpen your pencil

Solution

Avant de regarder la page suivante, notez rapidement les trois choses que nous devons faire avec ce code pour l'adapter à notre framework :

1. implémenter l'interface du menu
2. se débarrasser de getItems()
3. ajoutez createIterator() et renvoyez un itérateur qui peut parcourir les valeurs HashMap



Solution Code Magnets

L'itérateur de menu Diner « Alternatif » déchiffré.

```
importer java.util.Iterator;
importer java.util.Calendar;
```

classe publique AlternatingDinerMenuItemIterator

implémente Iterator<MenuItem>

```
MenuItem[] éléments;
position int;
```

public AlternatingDinerMenuItemIterator (éléments MenuItem []) {

```
this.items = éléments;
position = Calendrier.JOUR_DE_LA_SEMAINE % 2;
```

}

public booléen hasNext() {

```
si (position >= éléments.length || éléments[position] == null) {
    retourner faux;
} autre {
    renvoie vrai ;
}
```

}

élément de menu public suivant() {

```
MenuItem menuItem = éléments[position];
position = position + 2;
retourner menuItem;
```

}

public void remove() {

lancer une nouvelle UnsupportedOperationException(
 "L'itérateur de menu alternatif du restaurant ne prend pas en charge remove()");

}

}

Notez que cette
implémentation d'Iterator ne
prend pas en charge remove().

* WHO DOES WHAT? *

SolutionUTION

Associez chaque motif à sa description :

Modèle

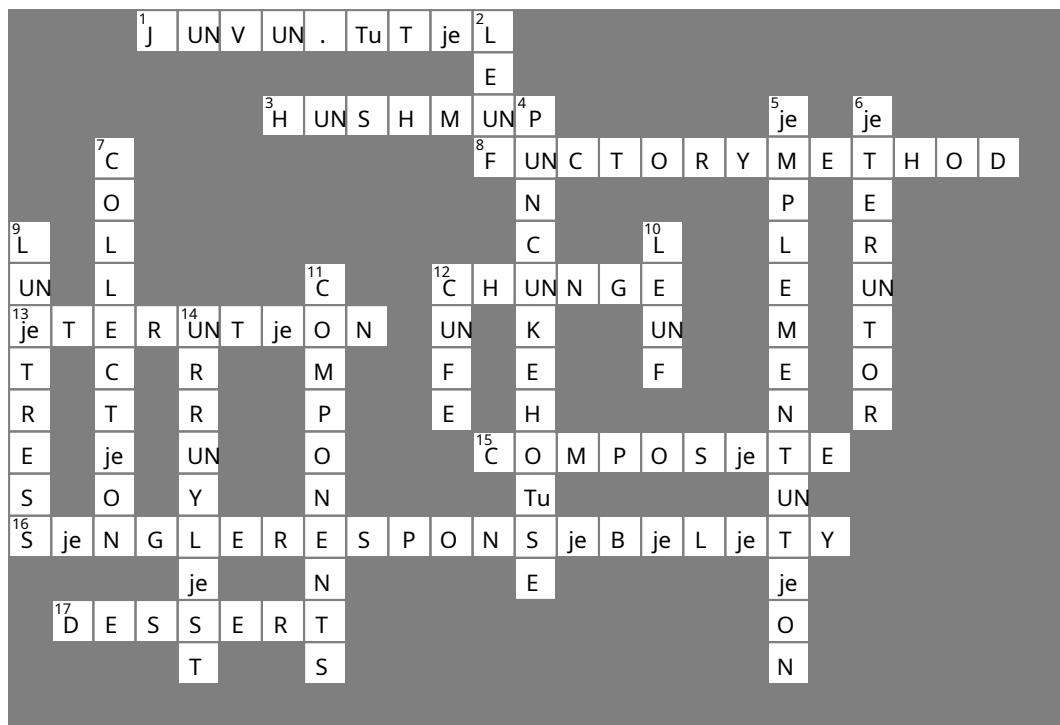
Description

Stratégie	Les clients traitent les collections d'objets et les objets individuels de manière uniforme
Adaptateur	Fournit un moyen de parcourir une collection d'objets sans exposer le implémentation de la collection
Itérateur	Simplifie l'interface d'un groupe de classes
Façade	Modifie l'interface d'une ou plusieurs classes
Composite	Permet à un groupe d'objets d'être notifié lorsque certains états changent
Observateur	Encapsule les comportements interchangeables et utilise la délégation pour décider lequel utiliser



Solution de mots croisés sur les modèles de conception

Réfléchissez bien à ce mot croisé composite. Voici notre solution.



10 Le modèle d'État

L'état des choses



Je pensais que les choses à Objectville étaient va être c'est si facile, mais maintenant à chaque faire demi-tour fois qu'il y a une autre demande de en entrant. Je changement, je suis au point de rupture ! peut-être que je sh Oh, j'aurais dû aller chez Betty Mercredi les motifs nocturnes se regroupent tout au Je suis dans une telle situat long, un état !

Un fait peu connu : les modèles de stratégie et d'état

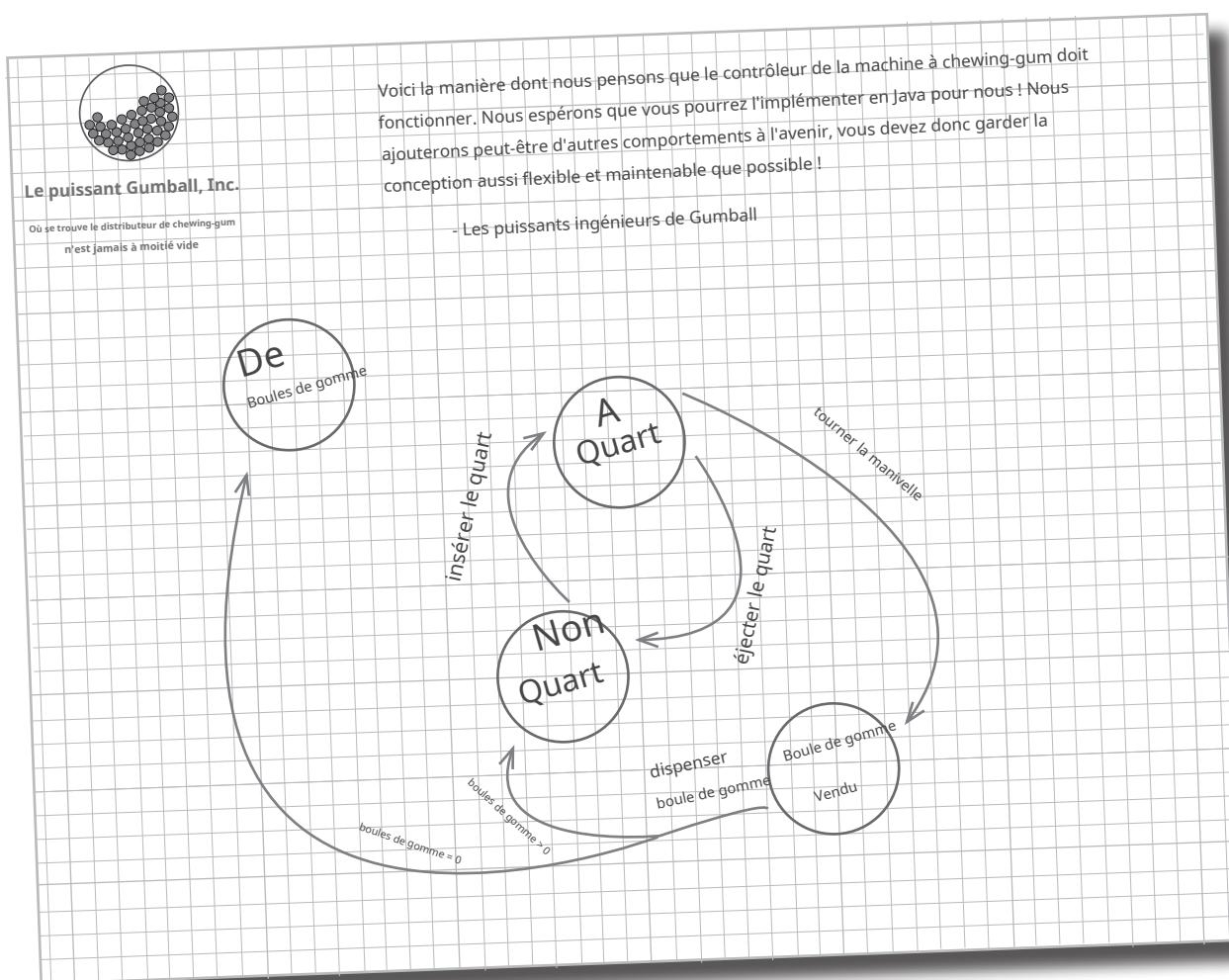
sont des jumeaux séparés à la naissance. On pourrait penser qu'ils mèneraient une vie similaire, mais le modèle de stratégie a donné naissance à une entreprise extrêmement prospère autour d'algorithmes interchangeables, tandis que l'État a pris le chemin peut-être plus noble d'aider les objets à contrôler leur comportement en modifiant leur état interne. Cependant, aussi différents que soient leurs chemins, vous trouverez sous-jacent presque exactement la même conception. Comment est-ce possible ? Comme vous le verrez, la stratégie et l'État ont des intentions très différentes. Commençons par creuser et voyons ce qu'est le modèle d'État, puis nous reviendrons pour explorer leur relation à la fin du chapitre.

virginie
~~z~~ Casse-mâchoires

Les distributeurs de bonbons sont devenus de plus en plus high-tech. En effet, les principaux fabricants ont découvert qu'en intégrant des processeurs dans leurs distributeurs de bonbons, ils peuvent augmenter leurs ventes, surveiller les stocks sur le réseau et mesurer la satisfaction des clients avec plus de précision.

Mais ces fabricants sont des experts en distributeurs de chewing-gum, pas des développeurs de logiciels, et ils ont demandé votre aide :

Du moins, c'est leur histoire - nous pensons qu'ils viennent de recevoir lassés de la technologie des années 1800, ils avaient besoin de trouver un moyen de rendre leur travail plus passionnant.



Conversation en cabine



Judy : Ce diagramme ressemble à un diagramme d'état.

Joe : Bien, chacun de ces cercles est un état...

Judy : ...et chacune des flèches est une transition d'état.

Franc: Ralentissez, vous deux, ça fait trop longtemps que je n'ai pas étudié les diagrammes d'état. Pouvez-vous me rappeler à quoi ils correspondent ?

Judy : Bien sûr, Frank. Regarde les cercles, ce sont des états. « Pas de pièce de 25 cents » est probablement l'état de départ du distributeur de chewing-gum, car il attend simplement que tu y mettes ta pièce de 25 cents. Tous les états ne sont que des configurations différentes de la machine qui se comportent d'une certaine manière et nécessitent une action pour passer à un autre état.

Joe : D'accord. Voyez-vous, pour aller dans un autre État, vous devez faire quelque chose comme mettre une pièce de 25 cents dans la machine. Vous voyez la flèche allant de « Pas de pièce de 25 cents » à « A une pièce de 25 cents » ?

Franc:Oui...

Joe :Cela signifie simplement que si le distributeur de chewing-gum est dans l'état « Pas de pièce de 25 cents » et que vous y mettez une pièce de 25 cents, il passera à l'état « Contient une pièce de 25 cents ». C'est la transition d'état.

Franc:Oh, je vois ! Et si je suis dans l'état « A 25 cents », je peux tourner la manivelle et passer à l'état « Gumball vendu », ou éjecter la pièce de 25 cents et revenir à l'état « Pas de pièce de 25 cents ».

Judy :Tu l'as compris !

Franc:Cela ne semble pas si mal alors. Nous avons évidemment quatre états, et je pense que nous avons également quatre actions : « insérer la pièce de 25 cents », « éjecter la pièce de 25 cents », « tourner la manivelle » et « distribuer ». Mais... lorsque nous distribuons, nous testons la présence de zéro ou plusieurs boules de gomme dans l'état « Boule de gomme vendue », puis nous passons soit à l'état « Plus de boules de gomme » soit à l'état « Pas de pièce de 25 cents ». Nous avons donc en fait cinq transitions d'un état à un autre.

Judy : Ce test pour zéro ou plus de boules de gomme implique également que nous devons également garder une trace du nombre de boules de gomme. Chaque fois que la machine vous donne une boule de gomme, il se peut que ce soit la dernière, et si c'est le cas, nous devons passer à l'état « Plus de boules de gomme ».

Joe :N'oubliez pas non plus que vous pourriez faire des choses absurdes, comme essayer d'éjecter la pièce de 25 cents lorsque le distributeur de chewing-gum est en mode « Pas de pièce de 25 cents », ou insérer deux pièces de 25 cents.

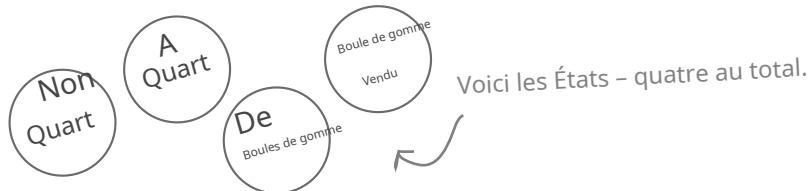
Franc:Oh, je n'y avais pas pensé ; nous devrons nous en occuper aussi.

Joe :Pour chaque action possible, il nous suffira de vérifier dans quel état nous nous trouvons et d'agir en conséquence. Nous pouvons le faire ! Commençons par mapper le diagramme d'état au code...

Machines d'état 101

Comment allons-nous passer de ce diagramme d'état au code réel ? Voici une brève introduction à l'implémentation des machines d'état :

- 1 Tout d'abord, rassemblez vos états :



- 2 Ensuite, créez une variable d'instance pour contenir l'état actuel et définissez des valeurs pour chacun des états :

Appelons simplement « Out of Gumballs » «
Sold Out » pour faire court.

```
final static int SOLD_OUT = 0; final static  
int NO_QUARTER = 1; final static int  
HAS_QUARTER = 2; final static int VENDU  
= 3;
```

int état = VENDU;

Voici chaque état représenté par
un entier unique...

... et voici une variable d'instance qui contient l'état actuel.
Nous allons continuer et la définir sur « Épuisé » puisque
la machine sera vide lorsqu'elle sera sortie de sa boîte et
mise sous tension pour la première fois.

- 3 Nous rassemblons maintenant toutes les actions qui peuvent se produire dans le système :

insérer le quart

tourner la manivelle

éjecter le quart

dispenser

Ces actions sont

le distributeur de boules de gomme
interface — les choses que
vous pouvez faire avec elle.

En regardant le diagramme, l'invocation de l'une de
ces actions provoque une transition d'état.

La distribution est davantage une action interne
que la machine invoque sur elle-même.

- 4** Nous créons maintenant une classe qui agit comme machine d'état. Pour chaque action, nous créons une méthode qui utilise des instructions conditionnelles pour déterminer le comportement approprié dans chaque état. Par exemple, pour l'action « insérer un quart », nous pourrions écrire une méthode comme celle-ci :

```

public void insertQuarter() {

    si (état == A_QUARTER) {
        System.out.println("Vous ne pouvez pas insérer un autre quart");

    } sinon si (état == NO_QUARTER) {
        état = HAS_QUARTER;
        System.out.println("Vous avez inséré un quart");

    } else if (état == VENDU) {
        System.out.println("Vous ne pouvez pas insérer un quart, la machine est en rupture de stock");

    } else if (état == VENDU) {
        System.out.println("Veuillez patienter, nous vous donnons déjà une boule de gomme");

    }
}

```

Chaque possible
l'état est vérifié
avec une condition
déclaration...

... et présente le comportement
approprié pour chaque état possible...

... mais peut également passer à d'autres états,
comme illustré dans le diagramme.

Ici nous parlons
à propos d'une technique courante : la
modélisation de l'état dans un objet en créant
une variable d'instance pour contenir les valeurs
d'état et en écrivant du code conditionnel dans
nos méthodes pour gérer
les différents états.



Avec ce rapide aperçu, allons mettre en œuvre la machine à boules de gomme !

Écrire le code

Il est temps d'implémenter la machine à chewing-gum. Nous savons que nous allons avoir une variable d'instance qui contient l'état actuel. À partir de là, nous devons simplement gérer toutes les actions, comportements et transitions d'état qui peuvent se produire. Pour les actions, nous devons implémenter l'insertion d'une pièce de 25 cents, le retrait d'une pièce de 25 cents, la rotation de la manivelle et la distribution d'une boule de chewing-gum ; nous avons également la condition de machine à chewing-gum vide à implémenter.

classe publique GumballMachine {

```
final static int SOLD_OUT = 0; final static
int NO_QUARTER = 1; final static int
HAS_QUARTER = 2; final static int VENDU
= 3;
```

```
int état = VENDU; int
nombre = 0;
```

```
machine à sous publique (nombre d'int) {
    ceci.count = compter;
    si (nombre > 0) {
        état = NO_QUARTER;
    }
}
```

```
public void insertQuarter() {
    si (état == A_QUARTER) {
        System.out.println("Vous ne pouvez pas insérer un autre quart");
    } else if (état == NO_QUARTER) {
        état = HAS_QUARTER;
        System.out.println("Vous avez inséré un quart");
    } else if (état == SOLD_OUT) {
        System.out.println("Vous ne pouvez pas insérer un quart, la machine est en rupture de
stock");
    } else if (état == SOLD) {
        System.out.println("Veuillez patienter, nous vous donnons déjà une boule de gomme");
    }
}
```

Voici les quatre états ; ils correspondent aux états du diagramme d'état de Mighty Gumball.

Voici la variable d'instance qui va suivre l'état actuel dans lequel nous nous trouvons. Nous commençons dans l'état SOLD_OUT.

Nous avons une deuxième variable d'instance qui garde une trace du nombre de boules de gomme dans la machine.

Le constructeur effectue un inventaire initial de boules de gomme. Si l'inventaire n'est pas nul, la machine entre dans l'état NO_QUARTER, ce qui signifie qu'elle attend que quelqu'un insère une pièce de 25 cents ; sinon, elle reste dans l'état SOLD_OUT.

Nous commençons maintenant à implémenter les actions en tant que méthodes....

Lorsqu'un quart est inséré...

... si un quart est déja inséré, nous le disons au client...

... sinon, nous acceptons le trimestre et passons à l'état HAS_QUARTER.

Si le client vient d'acheter une boule de gomme, il doit attendre que la transaction soit terminée avant d'insérer une autre pièce de 25 cents.

Et si la machine est épuisée, nous rejettions le quart.

```

public void ejectQuarter() {
    si (état == A_QUARTER) {
        System.out.println("Trimestre rendu"); état =
            NO_QUARTER;
    } sinon si (état == NO_QUARTER) {
        System.out.println("Vous n'avez pas inséré de quart"); } else if
        (state == VENDU) {
            System.out.println("Désolé, vous avez déjà tourné la manivelle"); } else if
        (state == SOLD_OUT) {
            System.out.println("Vous ne pouvez pas éjecter, vous n'avez pas encore inséré une pièce de 25 cents");
    }
}

// Maintenant, si le client essaie de retirer le quart...
... s'il y a un quart, on le renvoie et on revient à l'état NO_QUARTER...
... sinon, s'il n'y en a pas, on ne peut pas le rendre.

} Vous ne pouvez pas éjecter si la machine est épuisée, elle n'accepte pas les pièces de 25 cents !
Si le client vient de tourner la manivelle, nous ne pouvons pas effectuer de remboursement ; il a déjà la boule de gomme !

```

public void turnCrank()

```

    si (état == VENDU) {
        System.out.println("Se retourner deux fois ne vous rapportera pas une autre boule de gomme !");
    } else if (état == NO_QUARTER) {
        System.out.println("Vous avez tourné mais il n'y a pas de quart");
    } else if (état == SOLD_OUT) {
        System.out.println("Vous vous êtes retourné, mais il n'y a pas de boules de gomme");
    } else if (état == HAS_QUARTER) {
        System.out.println("Vous avez tourné...");
        état = VENDU;
        dispenser();
    }
}

// Quelqu'un essaie de tromper la machine.
// Nous avons besoin d'un premier quart.
// Nous ne pouvons pas livrer boules de gomme; il y a il n'y en a aucun.
// Succès ! Ils reçoivent une boule de gomme.
// Modifiez l'état sur VENDU et appelez la méthode dispense() de la machine.

```

public void dispense()

```

    si (état == VENDU) {
        System.out.println("Une boule de gomme sort de la fente"); count = count
            - 1;
        si (compte == 0) {
            System.out.println("Oups, plus de chewing-gum !");
            state = SOLD_OUT;
        } autre {
            état = NO_QUARTER;
        }
    } sinon si (état == NO_QUARTER) {
        System.out.println("Vous devez d'abord payer");
    } else if (état == SOLD_OUT) {
        System.out.println("Aucune boule de gomme n'a été distribuée");
    } else if (état == HAS_QUARTER) {
        System.out.println("Vous devez tourner la manivelle");
    }
}

// Appelé pour distribuer une boule de gomme.
// Nous sommes dans le état VENDU ; donner 'em une boule de gomme !
// Voici comment nous gérons la condition « plus de boules de gomme » :
// Si c'était le dernier, nous définissons l'état de la machine sur SOLD_OUT ; sinon, nous revenons à ne pas avoir de quart.
// Rien de tout cela ne devrait jamais arriver, mais si cela arrive, nous leur donnons une erreur, pas une boule de gomme.

```

// d'autres méthodes ici comme `toString()` et `refill()`

Tests en interne

Cela ressemble à une conception solide et soignée utilisant une méthodologie bien pensée, n'est-ce pas ? Faisons quelques tests en interne avant de le confier à Mighty Gumball pour qu'il soit chargé dans leurs véritables machines à boules de gomme. Voici notre harnais de test :

```
classe publique GumballMachineTestDrive {  
  
    public static void main(String[] args) {  
  
        Machine à boules de gomme Machine à boules de gomme = nouvelle Machine à boules de gomme(5);  
  
        Système.out.println(gumballMachine);  
  
        gumballMachine.insertQuarter();  
        gumballMachine.turnCrank();  
  
        Système.out.println(gumballMachine);  
  
        gumballMachine.insertQuarter();  
        gumballMachine.ejectQuarter();  
        gumballMachine.turnCrank();  
  
        Système.out.println(gumballMachine);  
  
        gumballMachine.insertQuarter();  
        gumballMachine.turnCrank();  
        gumballMachine.insertQuarter();  
        gumballMachine.turnCrank();  
        gumballMachine.ejectQuarter();  
  
        Système.out.println(gumballMachine);  
  
        gumballMachine.insertQuarter();  
        gumballMachine.insertQuarter();  
        gumballMachine.turnCrank();  
        gumballMachine.insertQuarter();  
        gumballMachine.turnCrank();  
        gumballMachine.insertQuarter();  
        gumballMachine.turnCrank();  
  
        Système.out.println(gumballMachine);  
    }  
}
```

The annotations describe the following sequence of events:

- Initial state: "Chargez-le avec cinq boules de gomme au total."
- First insertion: "Jetez une pièce de 25 cents dans..."
- First crank: "Tournez la manivelle, nous devrions avoir notre boule de gomme."
- Second insertion: "Jetez une pièce de 25 cents..."
- Second crank: "Demandez-la en retour."
- Third insertion: "Tournez la manivelle, nous ne devrions pas recevoir notre boule de gomme."
- Fourth insertion: "Jetez une pièce de 25 cents..."
- Fourth crank: "Tournez la manivelle, nous devrions récupérer notre boule de gomme."
- Fifth insertion: "Jetez-y une pièce de 25 cents..."
- Fifth crank: "Tournez la manivelle, nous devrions avoir notre boule de gomme."
- Sixth insertion: "Demandez un quart-arrière que nous n'avons pas mis."
- Sixth crank: "Imprimez à nouveau l'état de la machine."
- Seventh insertion: "Jetez DEUX pièces de 25 cents dans..."
- Seventh crank: "Tournez la manivelle, nous devrions avoir notre boule de gomme."
- Final note: "Passons maintenant aux tests de stress..." (with a thinking face emoji)
- Final print: "Imprimez l'état de cette machine une fois de plus."

Aide sur la fenêtre d'édition de fichiers mightygumball.com

```
%java GumballMachineTestDrive
Mighty Gumball, Inc.
Gumball debout compatible Java, modèle n° 2004
Inventaire : 5 gumballs
La machine attend le quart

Vous avez inséré un quart.
Vous avez tourné...
Une boule de gomme sort de la fente

Le puissant Gumball, Inc.
Gumball debout compatible Java Modèle #2004
Inventaire : 4 gumballs
La machine attend le quart

Vous avez inséré un quart Le
quart est revenu
Tu t'es retourné mais il n'y a pas de quartier

Le puissant Gumball, Inc.
Gumball debout compatible Java Modèle #2004
Inventaire : 4 gumballs
La machine attend le quart

Vous avez inséré un quart.
Vous avez tourné...
Une boule de gomme sort de la fente. Vous avez
inséré une pièce de 25 cents.
Tu t'es retourné...
Une boule de gomme sort de la fente. Vous
n'avez pas inséré une pièce de 25 cents.

Le puissant Gumball, Inc.
Gumball debout compatible Java, modèle n° 2004
Inventaire : 2 gumballs
La machine attend le quart

Vous avez inséré un quart
Tu ne peux pas insérer un autre quart
Tu as tourné...
Une boule de gomme sort de la fente. Vous avez
inséré une pièce de 25 cents.
Tu t'es retourné...
Une boule de gomme sort de la fente. Oups,
plus de boules de gomme !
Tu ne peux pas insérer un quart, la machine est en rupture de stock Tu t'es
retourné, mais il n'y a pas de boules de gomme

Le puissant Gumball, Inc.
Gumball debout compatible Java Modèle #2004
Inventaire : 0 gumballs
La machine est épuisée
```

Vous saviez que cela allait arriver... une demande de changement !

Mighty Gumball, Inc. a chargé votre code dans sa toute nouvelle machine et ses experts en assurance qualité le mettent à l'épreuve. Jusqu'à présent, tout semble aller pour le mieux de leur point de vue.

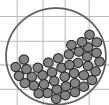
En fait, les choses se sont tellement bien passées qu'ils aimeraient passer au niveau supérieur...





Puzzle de conception

Dessinez un diagramme d'état pour un distributeur de boules de gomme qui gère le concours 1 sur 10. Dans ce concours, 10 % du temps, l'état Vendu conduit à la libération de deux boules, et non d'une seule. Vérifiez votre réponse avec la nôtre (à la fin du chapitre) pour vous assurer que nous sommes d'accord avant d'aller plus loin...



Le puissant Gumball, Inc.

Où se trouve le distributeur de chewing-gum
n'est jamais à moitié vide



Utilisez le papier à lettres de Mighty Gumball pour dessiner votre diagramme d'état.

L'ÉTAT désordonné des choses...

Ce n'est pas parce que vous avez écrit votre machine à chewing-gum en utilisant une méthodologie bien pensée qu'elle sera facile à étendre. En fait, lorsque vous revenez en arrière et regardez votre code et réfléchissez à ce que vous devrez faire pour le modifier, eh bien...

```
final static int SOLD_OUT = 0; final static  
int NO_QUARTER = 1; final static int  
HAS_QUARTER = 2; final static int VENDU  
= 3;
```

```
public void insertQuarter() {  
    // insérez le code du trimestre ici  
}
```

```
public void ejectQuarter() {  
    // éjecter le code du quart ici  
}
```

```
public void turnCrank() {  
    // tourner le code de la manivelle ici  
}
```

```
public void dispense() {  
    // distribuez le code ici  
}
```

Tout d'abord, vous devez ajouter un nouvel état GAGNANT ici. Ce n'est pas si mal...

... mais ensuite, vous devrez ajouter une nouvelle condition dans chaque méthode pour gérer l'état GAGNANT ; cela représente beaucoup de code à modifier.

turnCrank() deviendra particulièrement compliqué, car vous devrez ajouter du code pour vérifier si vous avez un GAGNANT, puis passer soit à l'état GAGNANT, soit à l'état VENDU.



Sharpen your pencil

Laquelle des propositions suivantes décrit l'état de notre mise en œuvre ? (Choisissez toutes les réponses appropriées.)

- A. Ce code n'adhère certainement pas au principe ouvert-fermé.
- B. Ce code rendrait fier un programmeur FORTRAN.
- C. Cette conception n'est même pas très orientée objet.
- D. Les transitions d'état ne sont pas explicites ; elles sont enfouies au milieu d'un tas d'instructions conditionnelles.
- E. Nous n'avons pas résumé ici ce qui varie.
- F. D'autres ajouts sont susceptibles de provoquer des bugs dans le code de travail.



Franc:Vous avez raison ! Nous devons refactoriser ce code pour qu'il soit facile à maintenir et à modifier.

Judy :Nous devrions vraiment essayer de localiser le comportement de chaque état afin que si nous apportons des modifications à un état, nous ne courions pas le risque de gâcher l'autre code.

Franc:Droite; dansEn d'autres termes, suivez ce vieux principe « encapsuler ce qui varie ».

Judy :Exactement.

Franc:Si nous plaçons le comportement de chaque État dans sa propre classe, alors chaque État met simplement en œuvre ses propres actions.

Judy :C'est vrai. Et peut-être que le distributeur de chewing-gum peut simplement déléguer à l'objet d'état qui représente l'état actuel.

Franc:Ah, tu es bon : privilégie la composition...plus de principes à l'œuvre.

Judy :Mignon. Bon, je ne suis pas sûr à 100 % de la façon dont cela va fonctionner, mais je pense que nous sommes sur la bonne voie.

Franc:Je me demande si cela facilitera l'ajout de nouveaux États ?

Judy :Je pense que oui... Nous devrons encore modifier le code, mais les changements seront beaucoup plus limités dans leur portée car l'ajout d'un nouvel état signifiera que nous devrons simplement ajouter une nouvelle classe et peut-être modifier quelques transitions ici et là.

Franc:J'aime bien cette idée. Commençons à élaborer ce nouveau design !

Le nouveau design

Il semble que nous ayons un nouveau plan : au lieu de maintenir notre code existant, nous allons le retravailler pour encapsuler les objets d'état dans leurs propres classes, puis déléguer à l'état actuel lorsqu'une action se produit.

Nous suivons ici nos principes de conception, nous devrions donc obtenir une conception plus facile à entretenir par la suite. Voici comment nous allons procéder :

- 1 Tout d'abord, nous allons définir une interface d'état qui contient une méthode pour chaque action dans la machine Gumball.**
- 2 Ensuite, nous allons implémenter une classe State pour chaque état de la machine. Ces classes seront responsables du comportement de la machine lorsqu'elle se trouve dans l'état correspondant.**
- 3 Enfin, nous allons nous débarrasser de tout notre code conditionnel et déléguer plutôt le travail à la classe State.**

Non seulement nous suivons les principes de conception, comme vous le verrez, nous implementons en fait le modèle d'état. Mais nous aborderons tous les aspects officiels du modèle d'état après avoir retravaillé notre code...



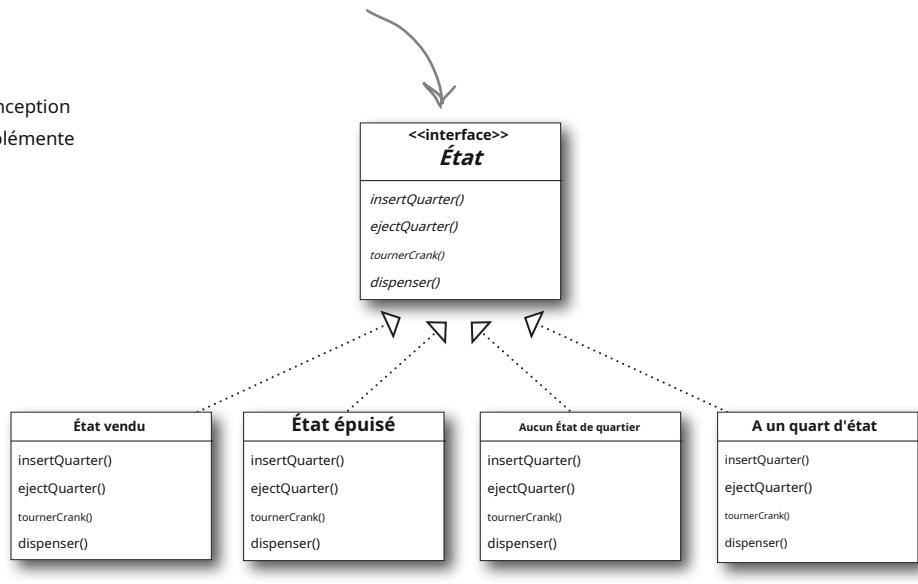
Définition des interfaces et des classes d'état

Commençons par créer une interface pour State, que tous nos états implémentent :

Voici l'interface pour tous les états. Les méthodes correspondent directement aux actions qui pourraient se produire sur le distributeur de chewing-gum (ce sont les mêmes méthodes que dans le code précédent).

Ensuite, prenez chaque état de notre conception et encapsulez-le dans une classe qui implémente l'interface State.

Pour déterminer les états dont nous avons besoin, nous regardons notre code précédent...



classe publique GumballMachine {

```

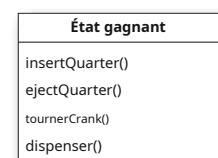
final static int SOLD_OUT = 0; final static
int NO_QUARTER = 1; final static int
HAS_QUARTER = 2; final static int VENDU
= 3;
  
```

```

int état = VENDU; int
nombre = 0;
  
```

... et nous mappons chaque état directement à une classe.

N'oubliez pas que nous avons également besoin d'un nouvel état « gagnant » qui implémente l'interface State. Nous y reviendrons après avoir réimplémenté la première version de Gumball Machine.

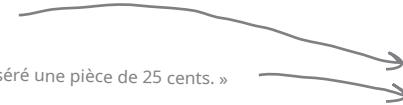


Sharpen your pencil



Pour implémenter nos états, nous devons d'abord spécifier le comportement des classes lorsque chaque action est appelée. Anotez le diagramme ci-dessous avec le comportement de chaque action dans chaque classe ; nous en avons déjà rempli quelques-unes pour vous.

Accédez à HasQuarterState.



Aucun État de quartier
insertQuarter()
ejectQuarter()
tournerCrank()
dispenser()

Dites au client : « Vous n'avez pas inséré une pièce de 25 cents. »

Accédez à SoldState.



A un quart d'état
insertQuarter()
ejectQuarter()
tournerCrank()
dispenser()

Dites au client : « Veuillez patienter, nous vous donnons déjà une boule de gomme. »



État vendu
insertQuarter()
ejectQuarter()
tournerCrank()
dispenser()

Distribuez une boule de gomme. Vérifiez le nombre de boules de gomme ; si
> 0, passez à NoQuarterState ; sinon, passez à SoldOutState.



État épuisé
insertQuarter()
ejectQuarter()
tournerCrank()
dispenser()

Dites au client : « Il n'y a pas de boules de gomme. »



État gagnant
insertQuarter()
ejectQuarter()
tournerCrank()
dispenser()

N'hésitez pas à remplir ce formulaire même si nous le mettrons en œuvre plus tard.



Mettre en œuvre nos cours d'État

Il est temps d'implémenter un état : nous savons quels comportements nous voulons ; il nous suffit de les mettre dans le code. Nous allons suivre de près le code de la machine à états que nous avons écrit, mais cette fois, tout est divisé en différentes classes.

Commençons par le NoQuarterState :

```

    Nous devons d'abord implémenter l'interface State.
    ↓
la classe publique NoQuarterState implémente State {
    Machine à boules de gomme Machine à boules de gomme;

    public NoQuarterState(GumballMachine gumballMachine) {
        ceci.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("Vous avez inséré un quart");
        gumballMachine.setState(gumballMachine.getHasQuarterState());
    }

    public void ejectQuarter() {
        System.out.println("Vous n'avez pas inséré de quart");
    }

    public void turnCrank() {
        System.out.println("Tu as tourné, mais il n'y a pas de quartier");
    }

    public void dispense() {
        System.out.println("Vous devez d'abord payer");
    }
}

```

Nous recevons une référence à la machine à boules de gomme via le constructeur. Nous allons simplement la stocker dans une variable d'instance.

Si quelqu'un insère un quart, nous imprimons un message indiquant que le quart a été accepté, puis nous changeons l'état de la machine en HasQuarterState.

Vous verrez comment cela fonctionne dans une seconde...

Vous ne pouvez pas récupérer votre argent si vous ne nous l'avez jamais donné !

Et vous ne pouvez pas obtenir de boule de gomme si vous ne nous payez pas.

Nous ne pouvons pas distribuer des boules de gomme sans paiement.



Ce que nous faisons, c'est mettre en œuvre les comportements qui sont appropriés à l'état dans lequel nous nous trouvons. Dans certains cas, ce comportement comprend le déplacement du distributeur de boules de gomme vers un nouvel état.

Refonte du distributeur de chewing-gum

Avant de terminer les classes d'état, nous allons retravailler la machine à boules de gomme, de cette façon, vous pourrez voir comment tout cela s'articule. Nous commencerons par les variables d'instance liées à l'état et passerons du code utilisant des entiers à celui utilisant des objets d'état :



Maintenant, regardons la classe GumballMachine complète...

```
classe publique GumballMachine {
```

```
    État venduOutState;
    État noQuarterState;
    L'État aQuarterState;
    État venduÉtat;
```

```
    État état;
    int nombre = 0;
```

```
    public GumballMachine(int nombreGumballs) {
        soldOutState = nouveau SoldOutState(this);
        noQuarterState = nouveau NoQuarterState(this);
        hasQuarterState = nouveau HasQuarterState(this);
        soldState = nouveau SoldState(this);
```

```
        this.count = nombreGumballs; si
        (nombreGumballs > 0) {
            état = noQuarterState;
        } sinon {
            état = État vendu ;
        }
```

```
    public void insertQuarter() {
        état.insertQuarter();
    }
    public void ejectQuarter() {
        état.ejectQuarter();
    }
    public void turnCrank() {
        état.turnCrank();
        état.dispense();
    }
```

```
    void setState(État état) {
        cet.état = état;
    }
```

```
    void releaseBall() {
        System.out.println("Une boule de gomme sort de la fente..."); if (count > 0) {
            compter = compter - 1;
        }
    }
    // Plus de méthodes ici, y compris des getters pour chaque état...
}
```

Voici à nouveau tous les États...

... et la variable d'instance State.

La variable d'instance count contient le nombre de boules de gomme — initialement, la machine est vide.

Notre constructeur prend le nombre initial de boules de gomme et le stocke dans une variable d'instance.
Il crée également les instances d'État, une de chaque.

S'il y a plus de 0 boules de gomme, nous définissons l'état sur NoQuarterState ; sinon, nous commençons dans SoldOutState.

Passons maintenant aux actions. Elles sont TRÈS FACILES à mettre en œuvre maintenant.
Il suffit de déléguer à l'état actuel.

Notez que nous n'avons pas besoin d'une méthode d'action pour dispense() dans GumballMachine car il s'agit simplement d'une action interne ; un utilisateur ne peut pas demander à la machine de distribuer directement. Mais nous appelons dispense() sur l'objet State à partir de la méthode turnCrank().

Cette méthode permet à d'autres objets (comme nos objets State) de faire passer la machine à un état différent.

La machine prend en charge une méthode d'assistance releaseBall() qui libère la balle et décrémente la variable d'instance count.

Cela inclut des méthodes telles que getNoQuarterState() pour obtenir chaque objet d'état et getCount() pour obtenir le nombre de boules de gomme.

Mettre en place davantage d'États

Maintenant que vous commencez à avoir une idée de la manière dont la machine à chewing-gum et les États s'articulent, implémentons les classes HasQuarterState et SoldState...

la classe publique HasQuarterState implémente State {

Machine à boules de gomme Machine à boules de gomme;

```
public HasQuarterState(GumballMachine gumballMachine) {
    ceci.gumballMachine = gumballMachine;
}

public void insertQuarter() {
    System.out.println("Vous ne pouvez pas insérer un autre quart");
}

public void ejectQuarter() {
    System.out.println("Trimestre retourné");
    gumballMachine.setState(gumballMachine.getNoQuarterState());
}

public void turnCrank() {
    System.out.println("Tu as tourné...");
    gumballMachine.setState(gumballMachine.getSoldState());
}

public void dispense() {
    System.out.println("Aucune boule de gomme n'a été distribuée");
}
```

Lorsque l'état est instancié, nous lui transmettons une référence à la GumballMachine. Cela permet de faire passer la machine à un état différent.

Un inapproprié action pour cela État.

Renvoyer le client quart et transition vers le NoQuarterState.

Lorsque la manivelle est tournée, nous faisons la transition la machine à l'état SoldState par en appelant son setState() méthode et en lui passant l'objet SoldState. L'objet SoldState est récupéré par getSoldState() méthode getter (il existe une de ces méthodes getter pour chaque état).

Un autre inapproprié action pour cela État.

Maintenant, regardons la classe SoldState...

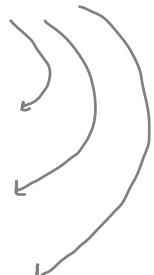
```
la classe publique SoldState implémente State {
    //variables constructeur et instance ici

    public void insertQuarter() {
        System.out.println("Veuillez patienter, nous vous donnons déjà une boule de gomme");
    }

    public void ejectQuarter() {
        System.out.println("Désolé, vous avez déjà tourné la manivelle");
    }

    public void turnCrank() {
        System.out.println("Tourner deux fois ne vous rapportera pas une autre boule de gomme !");
    }
```

Voici tous les
inappropriés
actions pour cela
État.

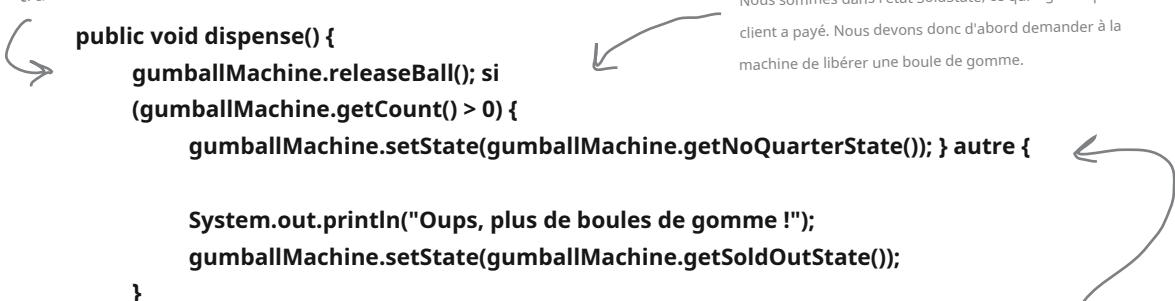


Et c'est ici que le vrai travail commence...

```
public void dispense() {
    gumballMachine.releaseBall(); si
    (gumballMachine.getCount() > 0 {
        gumballMachine.setState(gumballMachine.getNoQuarterState()); } autre {

            System.out.println("Oups, plus de boules de gomme !");
            gumballMachine.setState(gumballMachine.getSoldOutState());
        }
    }
```

Nous sommes dans l'état SoldState, ce qui signifie que le client a payé. Nous devons donc d'abord demander à la machine de libérer une boule de gomme.



Nous demandons ensuite à la machine quel est le nombre de boules de gomme et passons soit à l'état NoQuarterState, soit à l'état SoldOutState.



Revenons à l'implémentation de GumballMachine. Si la manivelle est tournée et qu'elle ne fonctionne pas (par exemple, si le client n'a pas inséré un quart au préalable), nous appelons quand même dispense(), même si ce n'est pas nécessaire. Comment pourriez-vous résoudre ce problème ?



Il nous reste une classe que nous n'avons pas implémentée : SoldOutState. Pourquoi ne pas l'implémenter ? Pour cela, réfléchissez bien à la façon dont le distributeur de chewing-gum doit se comporter dans chaque situation. Vérifiez votre réponse avant de passer à autre chose...

la classe publique SoldOutState implémente _____ {

Machine à boules de gomme Machine à boules de gomme;

public SoldOutState (GumballMachine gumballMachine) {

}

public void insertQuarter() {

}

public void ejectQuarter() {

}

public void turnCrank() {

}

public void dispense() {

}

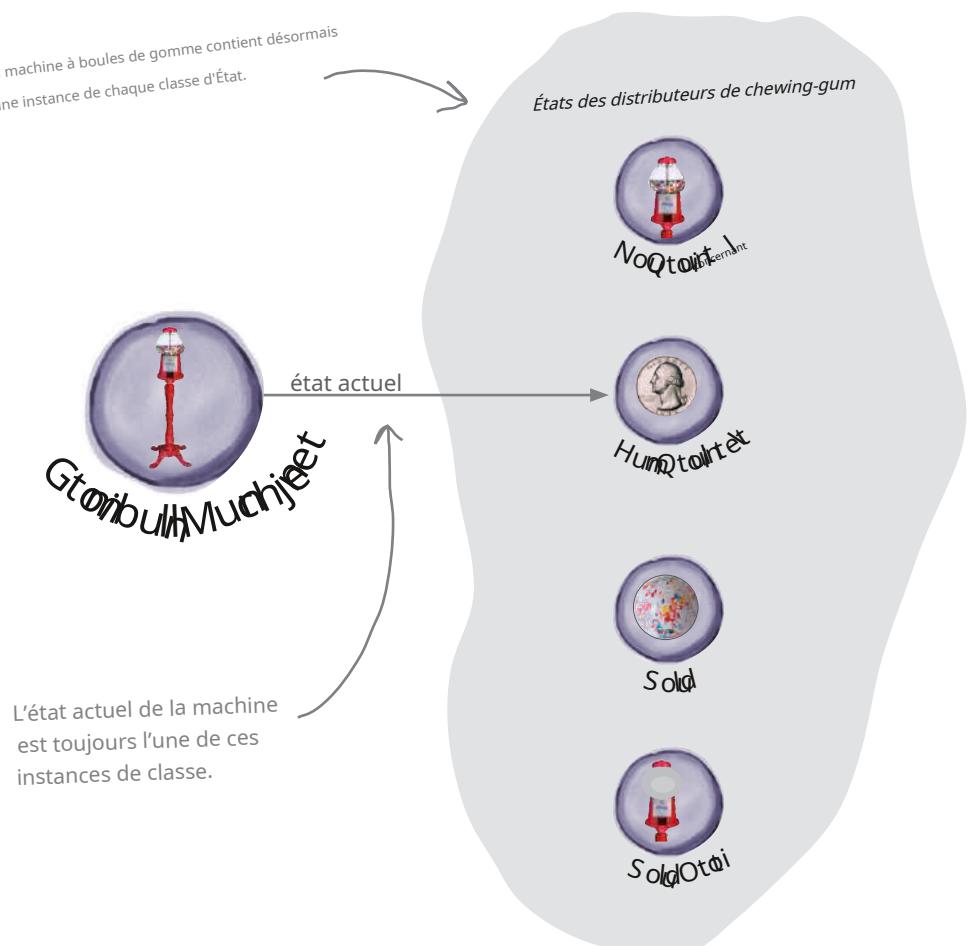
}

Jetons un œil à ce que nous avons fait jusqu'à présent...

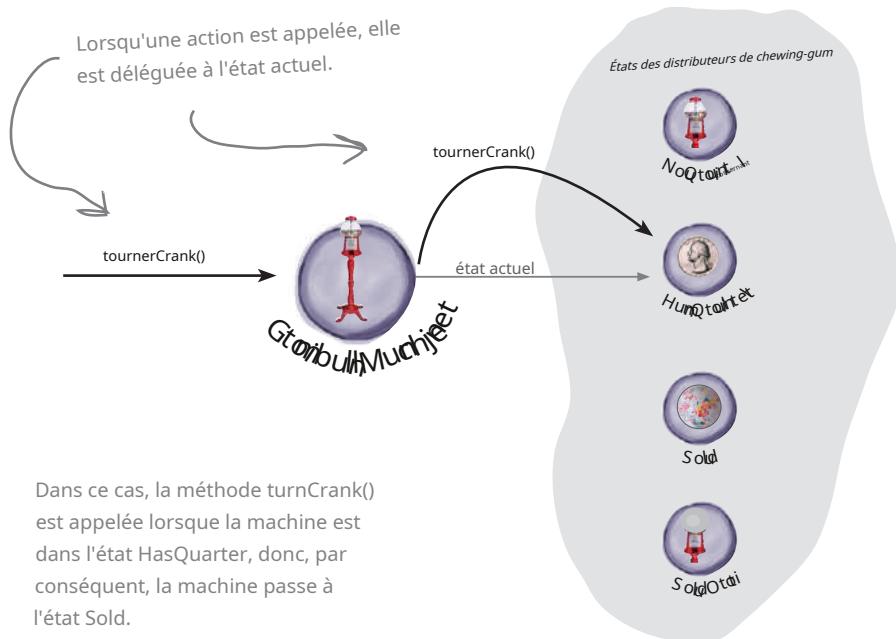
Pour commencer, vous disposez désormais d'une implémentation de Gumball Machine qui est *structurellement* assez différent de votre première version, et pourtant *fonctionnellement c'est exactement la même chose*. En modifiant structurellement l'implémentation, vous avez :

- ❖ Localisé le comportement de chaque état dans sa propre classe.
- ❖ Supprimé tous les problèmes des déclarations qu'il aurait été difficile de maintenir.
- ❖ J'ai fermé chaque état pour modification, et pourtant j'ai laissé la machine à gommes ouverte à l'extension en ajoutant de nouvelles classes d'état (et nous le ferons dans une seconde).
- ❖ J'ai créé une base de code et une structure de classe qui correspondent beaucoup plus étroitement au diagramme Mighty Gumball et qui sont plus faciles à lire et à comprendre.

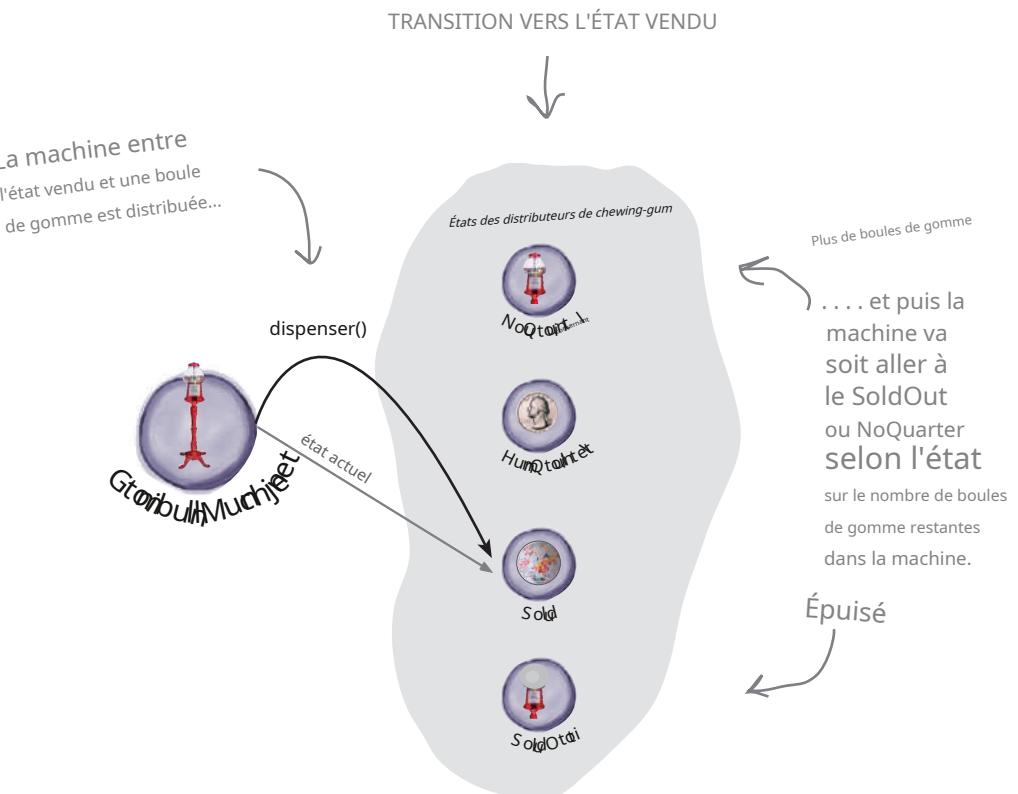
Regardons maintenant un peu plus l'aspect fonctionnel de ce que nous avons fait :



État transitions



Dans ce cas, la méthode turnCrank() est appelée lorsque la machine est dans l'état HasQuarter, donc, par conséquent, la machine passe à l'état Sold.





Dans les coulisses: Visite autoguidée



Suivez les étapes de la machine à boules de gomme en commençant par l'état NoQuarter. Annotez également le diagramme avec les actions et les résultats de la machine. Pour cet exercice, vous pouvez supposer qu'il y a beaucoup de boules de gomme dans la machine.



Le modèle d'État défini

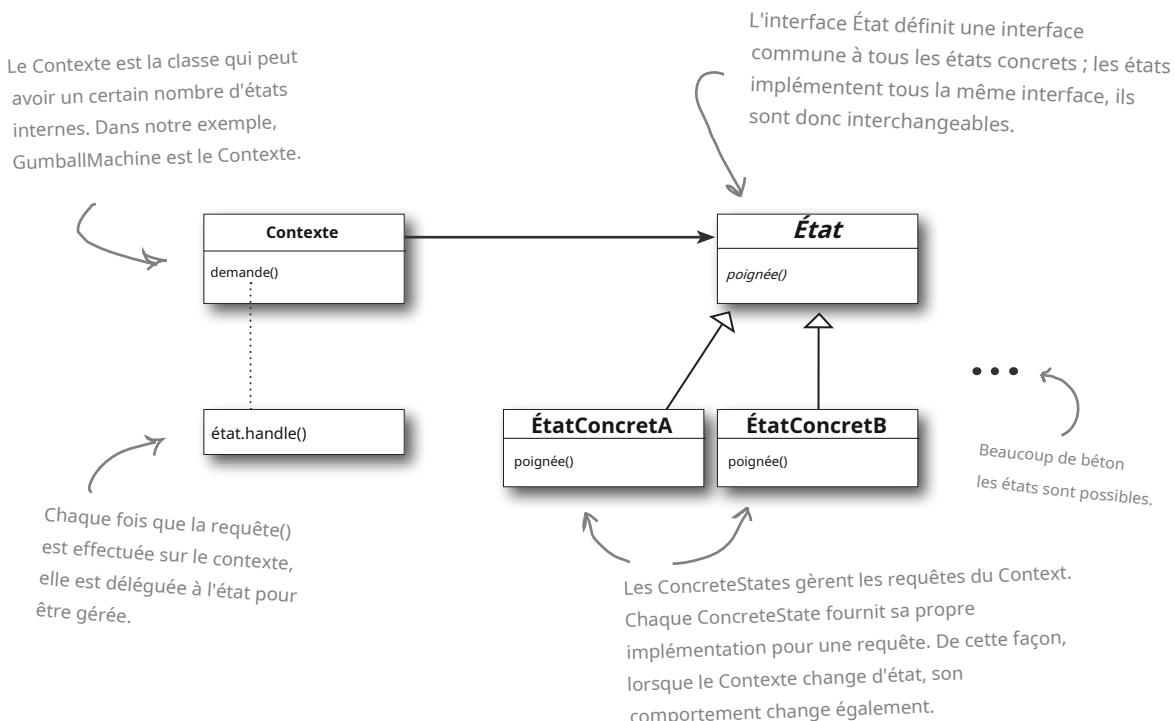
Oui, c'est vrai, nous venons d'implémenter le modèle d'état ! Voyons maintenant de quoi il s'agit :

Le modèle de l'État permet à un objet de modifier son comportement lorsque son état interne change. L'objet semblera changer de classe.

La première partie de cette description est très logique, n'est-ce pas ? Étant donné que le modèle encapsule l'état dans des classes distinctes et le délègue à l'objet représentant l'état actuel, nous savons que le comportement change en même temps que l'état interne. Le distributeur de chewing-gum fournit un bon exemple : lorsque le distributeur de chewing-gum est dans l'état NoQuarterState et que vous insérez une pièce de 25 cents, vous obtenez un comportement différent (la machine accepte la pièce de 25 cents) que si vous insérez une pièce de 25 cents alors qu'elle est dans l'état HasQuarterState (la machine rejette la pièce de 25 cents).

Qu'en est-il de la deuxième partie de la définition ? Que signifie pour un objet « sembler changer de classe » ? Pensez-y du point de vue d'un client : si un objet que vous utilisez peut complètement changer de comportement, alors il vous semble que l'objet est en fait instancié à partir d'une autre classe. En réalité, cependant, vous savez que nous utilisons la composition pour donner l'apparence d'un changement de classe en référençant simplement différents objets d'état.

Ok, il est maintenant temps de vérifier le diagramme de classe State Pattern :





Attendez une seconde ; d'après ce dont je me souviens du modèle de stratégie, ce diagramme de classe est EXACTEMENT le même.

Vous avez un bon œil (ou vous avez lu le début du chapitre) ! Oui, les diagrammes de classes sont essentiellement les mêmes, mais les deux modèles diffèrent dans leur *intention*.

Avec le modèle d'état, nous disposons d'un ensemble de comportements encapsulés dans des objets d'état ; à tout moment, le contexte délègue à l'un de ces états. Au fil du temps, l'état actuel change dans l'ensemble des objets d'état pour refléter l'état interne du contexte, de sorte que le comportement du contexte change également au fil du temps. Le client ne sait généralement que très peu de choses, voire rien, sur les objets d'état.

Avec Strategy, le client spécifie généralement l'objet de stratégie avec lequel le contexte est composé. Or, bien que le modèle offre la flexibilité de modifier l'objet de stratégie au moment de l'exécution, il existe souvent un objet de stratégie qui est le plus approprié pour un objet de contexte. Par exemple, dans le chapitre 1, certains de nos canards ont été configurés pour voler avec un comportement de vol typique (comme les canards colverts), tandis que d'autres ont été configurés avec un comportement de vol qui les maintient au sol (comme les canards en caoutchouc et les canards leurre).

En général, considérez le modèle Strategy comme une alternative flexible à la sous-classification ; si vous utilisez l'héritage pour définir le comportement d'une classe, vous êtes alors bloqué avec ce comportement même si vous devez le modifier. Avec Strategy, vous pouvez modifier le comportement en composant avec un objet différent.

Considérez le modèle d'état comme une alternative à la mise en place de nombreuses conditions dans votre contexte ; en encapsulant les comportements dans des objets d'état, vous pouvez simplement modifier l'objet d'état dans le contexte pour modifier son comportement.

there are no Dumb Questions

Q:

Dans GumballMachine, les états décident de l'état suivant. Les ConcreteStates décident-ils toujours de l'état suivant à atteindre ?

UN:

Non, pas toujours. L'alternative est de laisser le contexte décider du déroulement des transitions d'état.

En règle générale, lorsque les transitions d'état sont fixes, elles sont appropriées pour être placées dans le contexte ; cependant, lorsque les transitions sont plus dynamiques, elles sont généralement placées dans les classes d'état elles-mêmes (par exemple, dans GumballMachine, le choix de la transition vers NoQuarter ou SoldOut dépendait du nombre de gumballs d'exécution).

L'inconvénient d'avoir des transitions d'état dans les classes d'état est que nous créons des dépendances entre les classes d'état. Dans notre implémentation de GumballMachine, nous avons essayé de minimiser cela en utilisant des méthodes getter sur le contexte, plutôt que de coder en dur des classes d'état concrètes explicites.

Notez qu'en prenant cette décision, vous décidez quelles classes sont fermées à la modification (les classes de contexte ou d'état) à mesure que le système évolue.

Q:

Les clients interagissent-ils directement avec les États ?

UN:

Non. Les états sont utilisés par le contexte pour représenter son état interne et son comportement, donc toutes les requêtes adressées aux états proviennent du contexte. Les clients ne modifient pas directement l'état du contexte. C'est le travail du contexte de superviser son état, et vous ne souhaitez généralement pas qu'un client modifie l'état d'un contexte sans que ce contexte ne le sache.

Q:

Si j'ai de nombreuses instances du contexte dans mon application, est-il possible de partager les objets d'état entre elles ?

UN:

Oui, absolument, et en fait, il s'agit d'un scénario très courant. La seule exigence est que vos objets d'état ne conservent pas leur propre contexte interne ; sinon, vous auriez besoin d'une instance unique par contexte.

Pour partager vos états, vous devez généralement affecter chaque état à une variable d'instance statique. Si votre état doit utiliser des méthodes ou des variables d'instance dans votre contexte, vous devrez également lui donner une référence au contexte dans chaque méthode handler().

Q:

Il semble que l'utilisation du modèle d'état augmente toujours le nombre de classes dans nos conceptions. Regardez combien de classes de plus notre GumballMachine avait par rapport à la conception originale !

UN:

Vous avez raison ; en encapsulant le comportement de l'état dans des classes d'état distinctes, vous retrouvez toujours avec plus de classes dans votre conception. C'est souvent le prix à payer pour la flexibilité. À moins que votre code ne soit une implémentation « unique » que vous allez jeter (oui, c'est vrai), envisagez de la construire avec les classes supplémentaires et vous en remercierez probablement par la suite. Notez que souvent, ce qui est important, c'est le nombre de classes que vous exposez à vos clients, et il existe des moyens de cacher ces classes supplémentaires à vos clients (par exemple, en les déclarant privées du package).

Envisagez également l'alternative : si vous avez une application qui a beaucoup d'états et que vous décidez de ne pas utiliser d'objets séparés, vous retrouvez avec des instructions conditionnelles très volumineuses et monolithiques. Cela rend votre code difficile à maintenir et à comprendre. En utilisant des objets, vous rendez les états explicites et réduisez l'effort nécessaire pour comprendre et maintenir votre code.

Q:

Le diagramme de classe State Pattern montre que State est une classe abstraite. Mais n'avez-vous pas utilisé une interface dans l'implémentation de l'état du distributeur de chewing-gum ?

UN:

Oui. Étant donné que nous n'avions aucune fonctionnalité commune à intégrer dans une classe abstraite, nous avons opté pour une interface. Dans votre propre implémentation, vous souhaiterez peut-être envisager une classe abstraite. Cela présente l'avantage de vous permettre d'ajouter ultérieurement des méthodes à la classe abstraite, sans interrompre les implémentations d'état concrètes.

Il nous reste encore à terminer le jeu Gumball 1 en 10

N'oubliez pas que nous n'avons pas encore terminé. Nous devons implémenter un jeu, mais maintenant que nous avons implémenté le modèle d'état, cela devrait être un jeu d'enfant. Tout d'abord, nous devons ajouter un état à la classe GumballMachine :

```
classe publique GumballMachine {
```

```
    État venduOutState;
    État noQuarterState;
    L'État aQuarterState;
    État venduÉtat;
    État vainqueurÉtat;
```

Tout ce que vous devez ajouter ici est le nouveau WinnerState et l'initialiser dans le constructeur.

```
    État état = soldOutState; int count
    = 0;
    // méthodes ici
```

N'oubliez pas que vous devez également ajouter une méthode getter pour WinnerState.

```
}
```

Implémentons maintenant la classe WinnerState ; elle est remarquablement similaire à la classe SoldState :

```
la classe publique WinnerState implémente State {
```

```
// variables d'instance et constructeur //
message d'erreur insertQuarter // message
d'erreur ejectQuarter
// message d'erreur turnCrank
```

Tout comme SoldState.

```
public void dispense() {
    gumballMachine.releaseBall();
    si (gumballMachine.getCount() == 0) {
        gumballMachine.setState(gumballMachine.getSoldOutState()); } autre {
```

Ici, nous libérons deux boules de gomme, puis nous passons soit au NoQuarterState, soit au SoldOutState.

```
        gumballMachine.releaseBall();
        System.out.println("VOUS ÊTES GAGNANT ! Vous avez reçu deux boules de gomme pour votre pièce de 25
cents"); if (gumballMachine.getCount() > 0) {
```

Si nous avons une deuxième boule de gomme, nous la libérons.

```
            gumballMachine.setState(gumballMachine.getNoQuarterState()); } autre {
```

```
                System.out.println("Oups, plus de boules de gomme !");
                gumballMachine.setState(gumballMachine.getSoldOutState());
            }
        }
```

Si nous étions capables pour libérer deux boules de gomme, nous laissons l'utilisateur sait il était un gagnant.

```
}
```

Finir le jeu

Il ne nous reste plus qu'un changement à faire : nous devons implémenter le jeu de hasard et ajouter une transition au WinnerState. Nous allons ajouter les deux à HasQuarterState puisque c'est là que le client tourne la manivelle :

```
la classe publique HasQuarterState implémente State {  
    Aléatoire randomWinner = new Random(System.currentTimeMillis());  
  
    Machine à boules de gomme Machine à boules de gomme;  
  
    public HasQuarterState(GumballMachine gumballMachine) {  
        ceci.gumballMachine = gumballMachine;  
    }  
  
    public void insertQuarter() {  
        System.out.println("Vous ne pouvez pas insérer un autre quart");  
    }  
  
    public void ejectQuarter() {  
        System.out.println("Trimestre retourné");  
        gumballMachine.setState(gumballMachine.getNoQuarterState());  
    }  
  
    public void turnCrank() {  
        System.out.println("Vous avez tourné..."); int  
        winner = randomWinner.nextInt(10);  
        si ((gagnant == 0) && (gumballMachine.getCount() > 1)) {  
            gumballMachine.setState(gumballMachine.getWinnerState()); } autre {  
            gumballMachine.setState(gumballMachine.getSoldState());  
        }  
    }  
  
    public void dispense() {  
        System.out.println("Aucune boule de gomme n'a été distribuée");  
    }  
}
```

Tout d'abord, nous ajoutons un nombre aléatoire générateur à générer les 10% chance de gagner...

... ensuite nous déterminons si ce client a gagné.

S'ils ont gagné et qu'il reste suffisamment de boules de gomme pour qu'ils en aient deux, nous allons à WinnerState ; sinon, nous allons à SoldState (comme nous l'avons toujours fait).

Wow, c'était assez simple à mettre en œuvre ! Nous avons simplement ajouté un nouvel état à la GumballMachine, puis nous l'avons implémenté. Il ne nous restait plus qu'à implémenter notre jeu de hasard et à passer à l'état correct. Il semble que notre nouvelle stratégie de code porte ses fruits...

Démo pour le PDG de Mighty Gumball, Inc.

Le PDG de Mighty Gumball est venu nous voir pour une démonstration du code de votre nouveau jeu de gumball. Espérons que tous ces états soient en ordre ! Nous garderons la démonstration courte et agréable (la courte durée d'attention des PDG est bien documentée), mais espérons-le suffisamment longue pour que nous gagnions au moins une fois.

```

class public GumballMachineTestDrive {
    public static void main(String[] args) {
        Machine à boules de gomme Machine à boules de gomme = nouvelle Machine à boules de gomme(5);

        Système.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        Système.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        Système.out.println(gumballMachine);
    }
}

Ce code n'a vraiment pas changé du tout ; nous l'avons juste un peu raccourci.

```

Encore une fois, commencez avec une machine à boules de gomme avec 5 boules de gomme.

Nous voulons obtenir un État gagnant, alors nous continuons à pomper dans ces quartiers et à faire tourner la manivelle. Nous imprimons l'état du distributeur de chewing-gum de temps en temps...

Toute l'équipe d'ingénierie attend à l'extérieur de la salle de conférence pour voir si la nouvelle conception basée sur le modèle d'état va fonctionner !!





```
Aide sur la fenêtre d'édition de fichier Whenisgumballajawbreaker ?  
%java GumballMachineTestDrive  
Mighty Gumball, Inc.  
Gumball debout compatible Java, modèle n° 2004  
Inventaire : 5 gumballs  
La machine attend le quart  
  
Vous avez inséré un quart.  
Vous avez tourné...  
Une boule de gomme sort de la fente... Une  
boule de gomme sort de la fente...  
VOUS ÊTES GAGNANT ! Vous avez reçu deux boules de gomme pour votre pièce de 25 cents  
  
Le puissant Gumball, Inc.  
Gumball debout compatible Java, modèle n° 2004  
Inventaire : 3 gumballs  
La machine attend le quart  
  
Vous avez inséré un quart.  
Vous avez tourné...  
Une boule de gomme sort de la fente... Vous avez  
inséré une pièce de 25 cents  
Tu t'es retourné...  
Une boule de gomme sort de la fente... Une  
boule de gomme sort de la fente...  
VOUS ÊTES GAGNANTS ! Vous avez reçu deux boules de gomme pour votre pièce de 25 cents  
Oups, plus de boules de gomme !  
  
Le puissant Gumball, Inc.  
Gumball debout compatible Java Modèle #2004  
Inventaire : 0 gumballs  
La machine est épuisée à %
```

there are no
Dumb Questions

Q: UN:

Pourquoi avons-nous besoin du WinnerState ? Ne pourrions-nous pas simplement demander au SoldState de distribuer deux boules de gomme ?

C'est une excellente question. SoldState et WinnerState sont presque identiques, sauf que WinnerState distribue deux boules de gomme au lieu d'une. Vous pourriez certainement mettre le code pour distribuer deux boules de gomme dans SoldState. L'inconvénient est, bien sûr, que vous avez maintenant DEUX états représentés dans une classe State : l'état dans lequel vous êtes gagnant et l'état dans lequel vous ne l'êtes pas. Vous sacrifiez donc la clarté de votre classe State pour réduire la duplication de code. Une autre chose à prendre en compte est le principe que vous avez appris dans le chapitre précédent : le principe de responsabilité unique. En plaçant la responsabilité WinnerState dans SoldState, vous venez de donner à SoldState DEUX responsabilités. Que se passe-t-il lorsque la promotion se termine ? Ou que les enjeux du concours changent ? Il s'agit donc d'un compromis qui se résume à une décision de conception.

Bravo ! Bon travail,
gang. Nos ventes explosent déjà avec le
nouveau jeu. Vous savez, nous fabriquons
également des machines à soda, et je pensais
que nous pourrions mettre
un de ces bras de machine à sous sur le
côté et en faire aussi un jeu. Nous avons
des enfants de quatre ans qui jouent avec les
distributeurs de chewing-gum ; pourquoi s'arrêter là ?



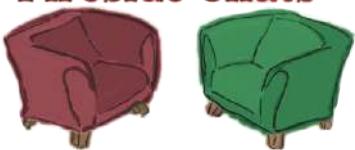
Vérification de la santé mentale...

Oui, le PDG de Mighty Gumball a probablement besoin d'un contrôle de santé mentale, mais ce n'est pas de cela dont nous parlons ici. Réfléchissons à certains aspects de la GumballMachine que nous pourrions vouloir consolider avant de lancer la version dorée :

- ❖ Nous avons beaucoup de codes dupliqués dans les états Vendu et Gagnant et nous souhaiterions peut-être les nettoyer. Comment procéder ? Nous pourrions transformer State en une classe abstraite et créer un comportement par défaut pour les méthodes ; après tout, les messages d'erreur tels que « Vous avez déjà inséré un quart » ne seront pas visibles par le client. Ainsi, tout le comportement de « réponse d'erreur » pourrait être générique et hérité de la classe abstraite State.
- ❖ La méthode dispense() est toujours appelée, même si la manivelle est tournée alors qu'il n'y a pas de quart. Bien que la machine fonctionne correctement et ne distribue pas à moins qu'elle ne soit dans le bon état, nous pourrions facilement résoudre ce problème en faisant en sorte que turnCrank() renvoie un booléen ou en introduisant des exceptions. Selon vous, quelle est la meilleure solution ?
- ❖ Toute l'intelligence des transitions d'état se trouve dans les classes d'état. Quels problèmes cela pourrait-il causer ? Voudrions-nous déplacer cette logique dans la GumballMachine ? Quels en seraient les avantages et les inconvénients ?
- ❖ Allez-vous instancier de nombreux objets GumballMachine ? Si tel est le cas, vous souhaiterez peut-être déplacer les instances d'état dans des variables d'instance statiques et les partager. Quels changements cela nécessiterait-il pour GumballMachine et les États ?

Bon sang Jim,
je suis une boule de gomme
machine, pas une
ordinateur!

Fireside Chats



La conférence de ce soir :**Une réunion de stratégie et de modèle d'État.**

Stratégie:

Hé, mec. Tu as entendu que j'étais au chapitre 1 ?

Je venais juste de donner un coup de main aux gars de Template Method : ils avaient besoin de moi pour les aider à terminer leur chapitre. Alors, de toute façon, que fait mon noble frère ?

Je ne sais pas, tu as toujours l'air de simplement copier ce que je fais et tu utilises des mots différents pour le décrire. Réfléchis-y : j'autorise les objets à intégrer différents comportements ou algorithmes grâce à la composition et à la délégation. Tu me copies simplement.

Ah oui ? Comment ça ? Je ne comprends pas.

Ouais, c'était quelque chose bien travailler... et je suis sûr que vous pouvez voir à quel point c'est plus puissant que d'hériter de votre comportement, n'est-ce pas ?

Désolé, vous allez devoir expliquer cela.

État:

Ouais, la nouvelle se répand clairement.

Comme d'habitude : aider les classes à présenter des comportements différents dans différents États.

J'admetts que ce que nous faisons est certainement lié, mais mon intention est totalement différente de la vôtre. Et la façon dont j'enseigne à mes clients à utiliser la composition et la délégation est totalement différente.

Eh bien, si vous passiez un peu plus de temps à penser à autre chose que *toi-même*, vous pourriez. Quoi qu'il en soit, réfléchissez à la façon dont vous travaillez : vous avez une classe que vous instanciez et vous lui donnez généralement un objet de stratégie qui implémente un comportement. Comme, dans le chapitre 1, vous distribuiez des comportements de cancan, n'est-ce pas ? Les vrais canards avaient un vrai cancan ; les canards en caoutchouc avaient un cancan qui couinait.

Oui, bien sûr. Maintenant, pensez à la façon dont je travaille, c'est totalement différent.

Stratégie:

Eh bien, allez, je peux aussi changer le comportement au moment de l'exécution ; c'est ça la composition !

Eh bien, je l'avoue, je n'encourage pas mes objets à avoir un ensemble bien défini de transitions entre les états. En fait, j'aime généralement contrôler la stratégie utilisée par mes objets.

Ouais, ouais, continue à vivre tes rêves, mon frère. Tu agis comme si tu étais un gros modèle comme moi, mais regarde ça : j'en suis au chapitre 1 ; ils t'ont coincé bien loin au chapitre 10. Je veux dire, combien de personnes vont réellement lire jusqu'ici ?

C'est mon frère, toujours le rêveur.

Etat:

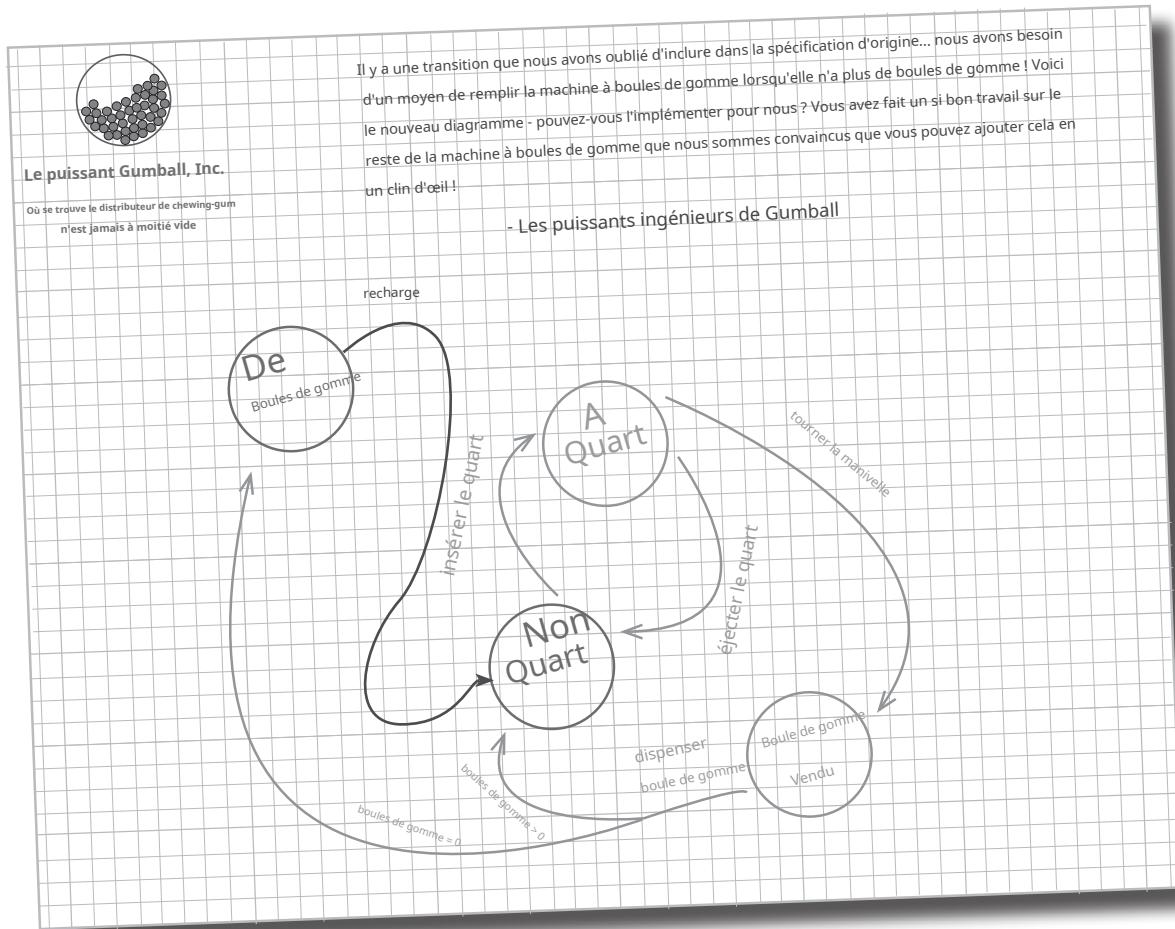
D'accord, lorsque mes objets Context sont créés, je peux leur indiquer l'état dans lequel démarrer, mais ils changent ensuite leur propre état au fil du temps.

Bien sûr que vous pouvez, mais ma façon de travailler est construite autour d'états discrets ; mes objets Context changent d'état au fil du temps en fonction de certaines transitions d'état bien définies. En d'autres termes, le changement de comportement est intégré à mon schéma — c'est ainsi que je travaille !

Nous avons déjà dit que nous avions la même structure, mais nos intentions sont très différentes. Admettons-le, le monde a une utilité pour nous deux.

Vous plaisantez ? C'est un livre de Head First et les lecteurs de Head First sont géniaux. Bien sûr qu'ils vont arriver au chapitre 10 !

On a presque oublié !





Sharpen your pencil

Nous avons besoin que vous écriviez la méthode refill() pour la machine à chewing-gum. Elle possède un argument (le nombre de chewing-gums que vous ajoutez à la machine) et doit mettre à jour le nombre de chewing-gums de la machine et réinitialiser l'état de la machine.

Tu as fait un travail incroyable ! J'ai d'autres idées qui vont changer l'industrie du chewing-gum et j'ai besoin que tu les mettes en pratique. Chuut ! Je te ferai part de ces idées dans le prochain chapitre.





Associez chaque motif à sa description :

Modèle	Description
État	Encapsulez les comportements interchangeables et utilisez la délégation pour décider quel comportement utiliser.
Stratégie	Les sous-classes décident comment pour implémenter des étapes dans un algorithme.
Méthode de modèle	Encapsuler le comportement basé sur l'état et déléguer comportement à l'état actuel.



Des outils pour votre boîte à outils de conception

C'est la fin d'un autre chapitre ; vous avez ici suffisamment de modèles pour réussir n'importe quel entretien d'embauche !

Principes OO

- Encapsuler ce qui varie.
- Privilégiez la composition plutôt que l'héritage.
- Programmez vers des interfaces, pas vers des implémentations.
- Efforcez-vous d'obtenir des conceptions faiblement couplées entre les objets qui interagissent.
- Les cours devraient être ouverts pour prolongation mais fermés pour modification.
- Dépendez des abstractions. Ne dépendez pas de classes concrètes.
- Parlez uniquement à vos amis. Ne nous appelez pas, nous vous appellerons.
- Une classe ne devrait avoir qu'une seule raison de changer.

Notions de base sur OO

- Abstraction
- Encapsulation
- Polymorphisme
- Héritage

Pas de nouveaux principes dans ce chapitre. Cela vous laisse le temps de réfléchir dessus.

Modèles OO

Si Oabtimpars exemple tetricamping-car et al définit une famille de classes qui peuvent être combinées pour créer une nouvelle famille de classes. En rapport avec ce, il existe plusieurs modèles de conception qui sont utilisés pour gérer l'état d'un système.

Exemple FT - Fournir un moyen de faire des nefs

En rapport avec ce, il existe plusieurs modèles de conception qui sont utilisés pour gérer l'état d'un système.

fonctionnalités UNtatdm.unobjet pointe à l'objet d'un autre

un classpadrafasfmefsonummoénormeumentempsdaeecnettc, i-

laitteEnnoientrcseatwbpoisituthlaehetedetisinfagerryeonquuest subclassreeqpsu.aerasatsmsaS enqeuounerbetuizzeecot-cl, liAtelonhlighgutuehlossetbdg, entchanges. supreopqrputaersbutaenmshd.eaoqvaueiborrluiiezewoohprcelnrlieonigrtisrsneiwnsq.tuitehsrndsa, lfafsntedart

supreopqrputaersbutaenmshd.eaoqvaueiborrluiiezewoohprcelnrlieonigrtisrsneiwnsq.tuitehsrndsa, lfafsntedart suppléants. opérations annulables.

Voici notre nouveau modèle. Si vous êtes gestion de l'État dans une classe, le modèle d'état vous donne une technique pour encapsulant cela État.



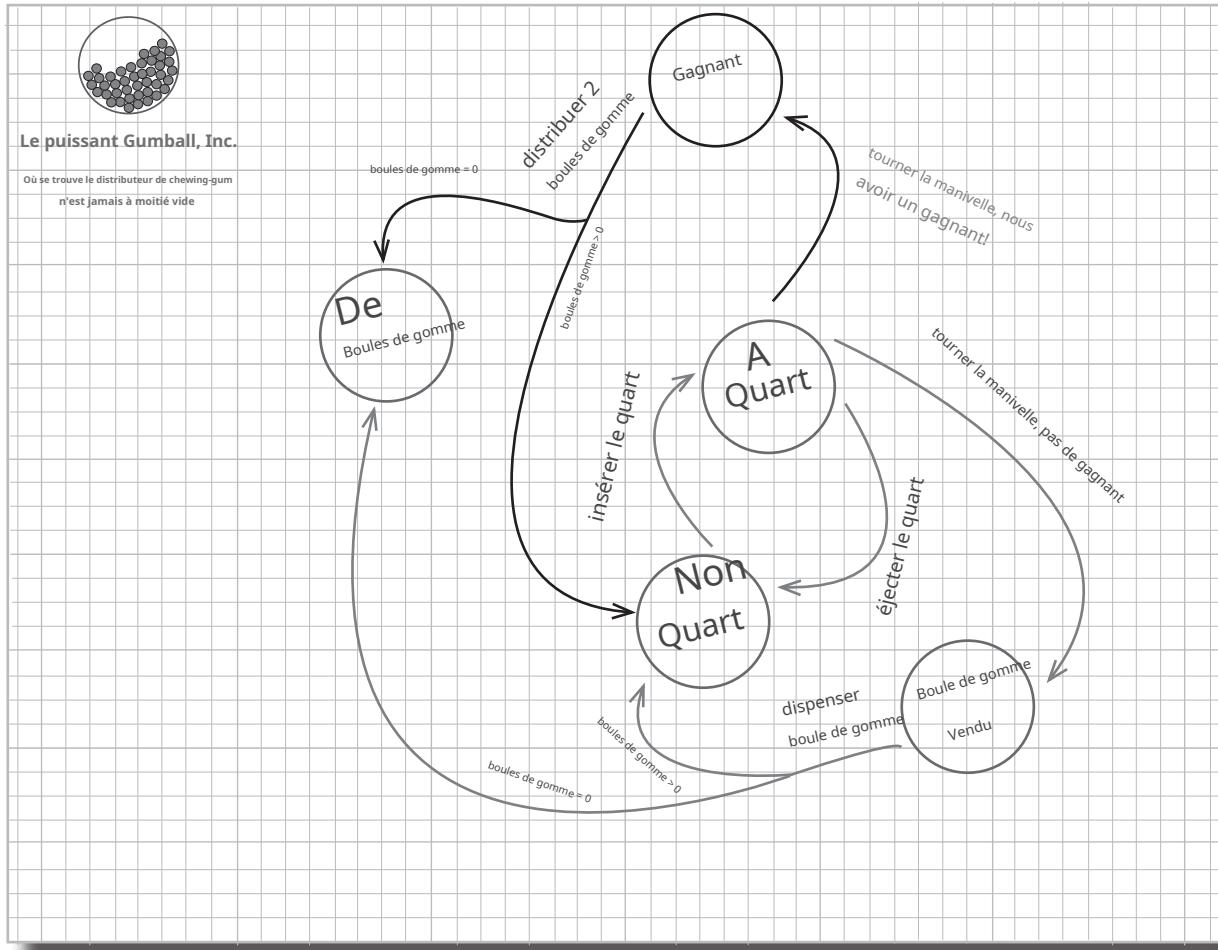
BULLET POINTS

- f Le modèle d'état permet à un objet d'avoir de nombreux comportements différents basés sur son état interne.
- f Contrairement à une machine à états procédurale, le modèle d'état représente chaque état comme une classe à part entière.
- f Le contexte obtient son comportement en déléguant à l'objet d'état actuel avec lequel il est composé.
- f En encapsulant chaque état dans une classe, nous localisons tous les changements qui devront être apportés.
- f Les modèles d'état et de stratégie ont le même diagramme de classes, mais ils diffèrent dans leur intention.
- f Le modèle de stratégie configure généralement les classes de contexte avec un comportement ou un algorithme.
- f Le modèle d'état permet à un contexte de modifier son comportement lorsque l'état du contexte change.
- f Les transitions d'état peuvent être contrôlées par les classes d'état ou par les classes de contexte.
- f L'utilisation du modèle d'état entraînera généralement un plus grand nombre de classes dans votre conception.
- f Les classes d'état peuvent être partagées entre les instances de contexte.



Solution du puzzle de conception

Dessinez un diagramme d'état pour un distributeur de boules de gomme qui gère le concours 1 sur 10. Dans ce concours, 10 % du temps, l'état Vendu conduit à la libération de deux boules, et non d'une seule. Voici notre solution.





Sharpen your pencil

Solution

Laquelle des propositions suivantes décrit l'état de notre implémentation ? (Choisissez toutes les réponses applicables.) Voici notre solution.

- A. Ce code n'adhère certainement pas au principe ouvert-fermé.
- B. Ce code rendrait fier un programmeur FORTRAN.
- C. Cette conception n'est même pas très orientée objet.
- D. Les transitions d'état ne sont pas explicites ; elles sont enfouies au milieu d'un tas d'instructions conditionnelles.
- E. Nous n'avons pas résumé ici ce qui varie.
- F. D'autres ajouts sont susceptibles de provoquer des bugs dans le code de travail.



Sharpen your pencil

Solution

Il nous reste une classe que nous n'avons pas implémentée : SoldOutState. Pourquoi ne pas l'implémenter ? Pour cela, réfléchissez bien à la façon dont le distributeur de chewing-gum doit se comporter dans chaque situation. Voici notre solution.

la classe publique SoldOutState implémente State {

Machine à boules de gomme Machine à boules de gomme;

```
public SoldOutState (GumballMachine gumballMachine) {
    ceci.gumballMachine = gumballMachine;
}

public void insertQuarter() {
    System.out.println("Vous ne pouvez pas insérer un quart, la machine est en rupture de stock");
}

public void ejectQuarter() {
    System.out.println("Vous ne pouvez pas éjecter, vous n'avez pas encore inséré une pièce de 25 cents");
}

public void turnCrank() {
    System.out.println("Vous vous êtes retourné, mais il n'y a pas de boules de gomme");
}

public void dispense() {
    System.out.println("Aucune boule de gomme n'a été distribuée");
}
```

Dans l'état épuisé, nous ne pouvons vraiment rien faire tant que quelqu'un n'a pas rempli le distributeur de chewing-gum.



Sharpen your pencil Solution

Pour implémenter les états, nous devons d'abord définir quel sera le comportement lorsque l'action correspondante est appelée. Annotez le diagramme ci-dessous avec le comportement de chaque action dans chaque classe ; voici notre solution.

Accédez à HasQuarterState.

Dites au client : « Vous n'avez pas inséré une pièce de 25 cents. »

Dites au client : « Vous vous êtes retourné, mais il n'y a pas de quartier. »

Dites au client : « Vous devez d'abord payer. »

Aucun État de quartier
insertQuarter()
ejectQuarter()
tournerCrank()
dispenser()

Dites au client : « Vous ne pouvez pas insérer un autre quart. »

Renvoyez le quart et passez à l'état NoQuarter.

Accédez à SoldState.

Dites au client : « Aucune boule de gomme n'a été distribuée. »

A un quart d'état
insertQuarter()
ejectQuarter()
tournerCrank()
dispenser()

Dites au client : « Veuillez patienter, nous vous donnons déjà une boule de gomme. »

Dites au client : « Désolé, vous avez déjà tourné la manivelle. »

Dites au client : « Si vous retournez deux fois, vous n'obtiendrez pas une autre boule de gomme. »

Distribuez une boule de gomme. Vérifiez le nombre de boules de gomme ; si > 0, passez à l'état NoQuarter ; sinon, passez à l'état SoldOut.

État vendu
insertQuarter()
ejectQuarter()
tournerCrank()
dispenser()

Dites au client : « La machine est en rupture de stock. »

Dites au client : « Vous n'avez pas encore inséré une pièce de 25 cents. »

Dites au client : « Il n'y a pas de boules de gomme. »

Dites au client : « Aucune boule de gomme n'a été distribuée. »

État épuisé
insertQuarter()
ejectQuarter()
tournerCrank()
dispenser()

Dites au client : « Veuillez patienter, nous vous donnons déjà une boule de gomme. »

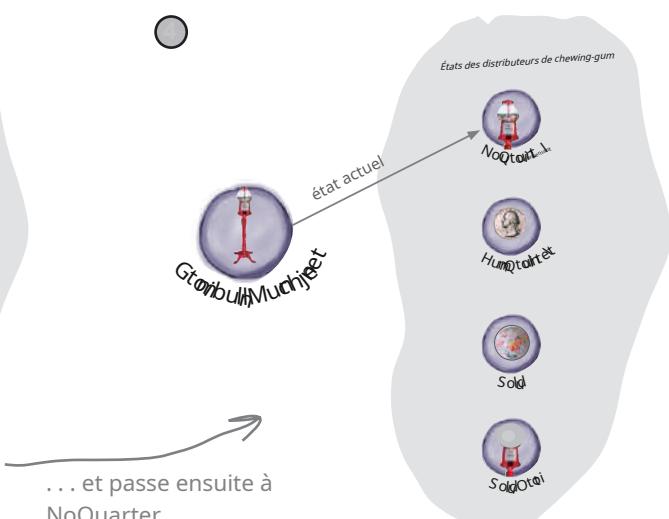
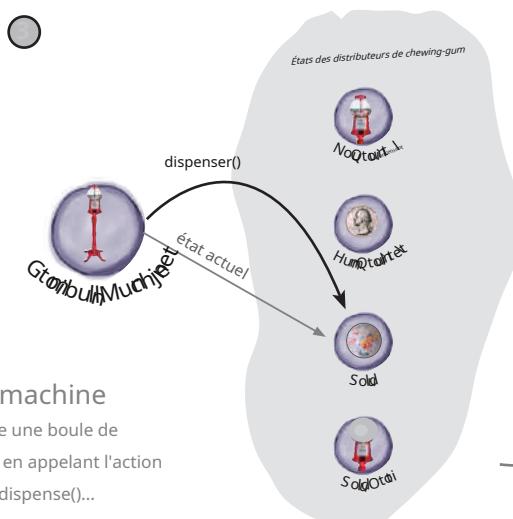
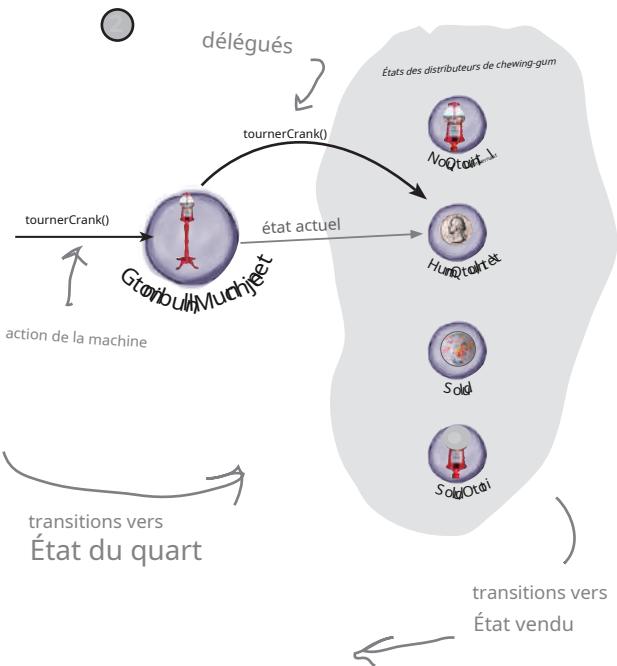
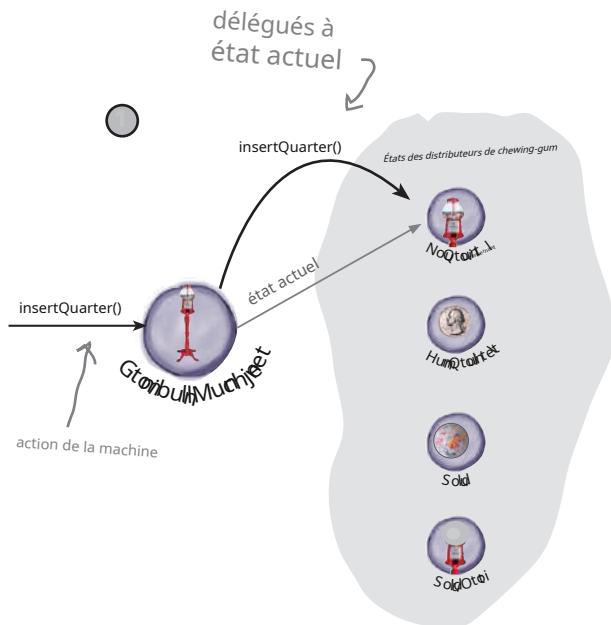
Dites au client : « Désolé, vous avez déjà tourné la manivelle. »

Dites au client : « Si vous retournez deux fois, vous n'obtiendrez pas une autre boule de gomme. »

Distribuez deux boules de gomme. Vérifiez le nombre de boules de gomme ; si > 0, passez à l'état NoQuarter ; sinon, passez à SoldOutState.

État gagnant
insertQuarter()
ejectQuarter()
tournerCrank()
dispenser()

Dans les coulisses: Visite autoguidée Solution





Associez chaque motif à sa description :

Modèle	Description
État	Encapsulez les comportements interchangeables et utilisez la délégation pour décider quel comportement utiliser.
Stratégie	Les sous-classes décident comment pour implémenter des étapes dans un algorithme.
Méthode de modèle	Encapsuler le comportement basé sur l'état et déléguer comportement à l'état actuel.



Sharpen your pencil

Solution

Pour remplir la machine à chewing-gum, nous ajoutons une méthode refill() à l'interface State, que chaque État doit implémenter. Dans tous les états, à l'exception de SoldOutState, la méthode ne fait rien. Dans SoldOutState, refill() passe à NoQuarterState. Nous ajoutons également une méthode refill() à GumballMachine qui augmente le nombre de chewing-gum, puis appelle la méthode refill() de l'état actuel.

```

public void recharge() {
    gumballMachine.setState(gumballMachine.getNoQuarterState());
}

void recharge(int nombre) {
    ceci.count += count;
    System.out.println("Le distributeur de chewing-gum vient d'être rempli ; son nouveau décompte est : " + this.count);
    state.refill();
}

```

← Nous ajoutons cette méthode à l'État épuisé...

11le modèle de proxy

Contrôler



Accès aux objets



Avec toi comme mandataire, je pourrai tripler le montant de l'argent du déjeuner que je peux extraire de mes amis !



Avez-vous déjà joué au bon flic, au méchant flic ?

Vous êtes le bon flic et vous fournissez tous vos services de manière agréable et amicale, mais vous ne voulez pas que tout le monde vous demande des services, alors vous laissez le mauvais flic contrôler l'accès à vous. C'est ce que font les proxies : contrôler et gérer l'accès. Comme vous allez le voir, il existe de nombreuses façons dont les proxies remplacent les objets qu'ils représentent. Les proxies sont connus pour transporter des appels de méthodes entiers sur Internet pour leurs objets mandatés ; ils sont également connus pour remplacer patiemment des objets assez paresseux.



Vous vous souvenez du PDG de Mighty Gumball, Inc. ?

Cela semble assez simple. Si vous vous en souvenez, nous avons déjà des méthodes dans le code de la machine à chewing-gum pour obtenir le nombre de boules de chewing-gum, `getCount()`, et pour obtenir l'état actuel de la machine, `getState()`.

Il nous suffit de créer un rapport qui peut être imprimé et renvoyé au PDG. Hmm, nous devrions probablement ajouter également un champ d'emplacement à chaque distributeur de chewing-gum ; de cette façon, le PDG peut garder les machines en ordre.

Allons-y et codons tout ça. Nous impressionnerons le PDG avec un délai d'exécution très rapide.

Codage du moniteur

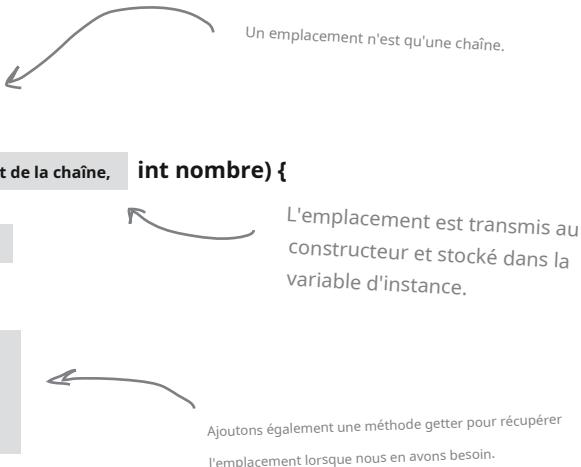
Commençons par ajouter le support à la classe GumballMachine afin qu'elle puisse gérer les emplacements :

```
classe publique GumballMachine {
    // autres variables d'instance
    Emplacement de la chaîne;

    machine à boules de gomme publique(Emplacement de la chaîne, int nombre) {
        // autre code constructeur ici
        cet.emplacement = emplacement;
    }

    chaîne publique getLocation() {
        lieu de retour;
    }

    // autres méthodes ici
}
```



Créons maintenant une autre classe, GumballMonitor, qui récupère l'emplacement de la machine, l'inventaire des boules de gomme et l'état actuel de la machine et les imprime dans un joli petit rapport :

```
classe publique GumballMonitor {
    Machine à boules de gomme;

    public GumballMonitor(machine GumballMachine) {
        cette.machine = machine;
    }

    rapport public void() {
        System.out.println("Distributeur de chewing-gum : " + machine.getLocation());
        System.out.println("Inventaire actuel : " + machine.getCount() + " chewing-gums");
        System.out.println("État actuel : " + machine.getState());
    }
}
```

Notre méthode report() imprime simplement un rapport avec l'emplacement, l'inventaire et l'état de la machine.

Le moniteur prend la machine dans son constructeur et l'affecte à la variable d'instance de la machine.

Test du moniteur

Nous avons mis cela en œuvre en un rien de temps. Le PDG va être ravi et étonné par nos compétences en développement.

Il ne nous reste plus qu'à instancier un GumballMonitor et lui donner une machine à surveiller :

```
classe publique GumballMachineTestDrive {
```

```
    public static void main(String[] args) {
        int nombre = 0;
```

Transmettez un emplacement et un nombre initial de boules de gomme sur la ligne de commande.

```
        si (args.length < 2) {
            System.out.println("GumballMachine <nom> <inventaire>");
            System.exit(1);
        }
```

N'oubliez pas de donner au constructeur un localisation et nombre...

```
        compte = Integer.parseInt(args[1]);
        GumballMachine gumballMachine = nouvelle GumballMachine(args[0], compte);
```

```
        moniteur GumballMonitor = nouveau GumballMonitor(gumballMachine);
```

... et instancier un moniteur et lui transmettre une machine pour fournir un rapport.

```
// reste du code de test ici
```

```
        surveiller.rapport();
```

Lorsque nous avons besoin d'un rapport sur la machine, nous appelons la méthode report().

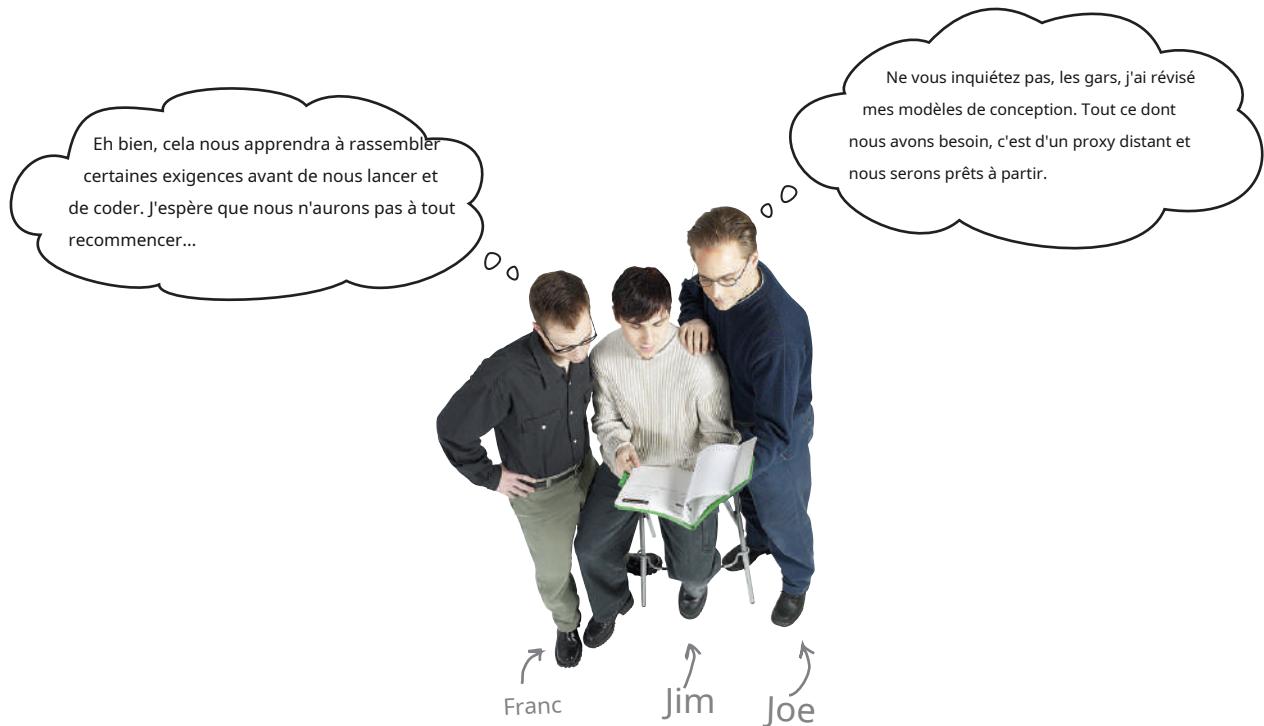
Fenêtre d'édition de fichier Aide FlyingFish

```
%java GumballMachineTestDrive Austin 112
Distributeur de chewing-gum : Austin
Inventaire actuel : 112 boules de gomme
État actuel : en attente du quart
```



La sortie du moniteur ressemble Super, mais je crois que je n'ai pas été clair. Je dois surveiller les distributeurs de chewing-gum À DISTANCE ! En fait, nous avons déjà mis en place les réseaux de surveillance. Allez les gars, vous êtes censés être la génération Internet !

Et voici le résultat !



Franc:Une télécommande quoi ?

Joe :Proxy distant. Pensez-y : nous avons déjà écrit le code du moniteur, n'est-ce pas ? Nous donnons à la classe GumballMonitor une référence à une machine et elle nous fournit un rapport. Le problème est que le moniteur s'exécute dans la même JVM que la machine à gommes et que le PDG veut s'asseoir à son bureau et surveiller les machines à distance ! Et si nous laissions notre classe GumballMonitor telle quelle, mais que nous lui donnions un proxy vers un objet distant ?

Franc:Je ne suis pas sûr de comprendre.

Jim :Moi non plus.

Joe :Commençons par le commencement... un proxy est un substitut d'un objet réel. Dans ce cas, le proxy agit comme s'il s'agissait d'un objet Gumball Machine, mais en coulisses, il communique via le réseau pour parler à la vraie GumballMachine distante.

Jim :Vous dites donc que nous gardons notre code tel quel et que nous donnons au moniteur une référence à une version proxy de GumballMachine...

Franc:Et ce proxy prétend être l'objet réel, mais en réalité, il communique simplement via le réseau avec l'objet réel.

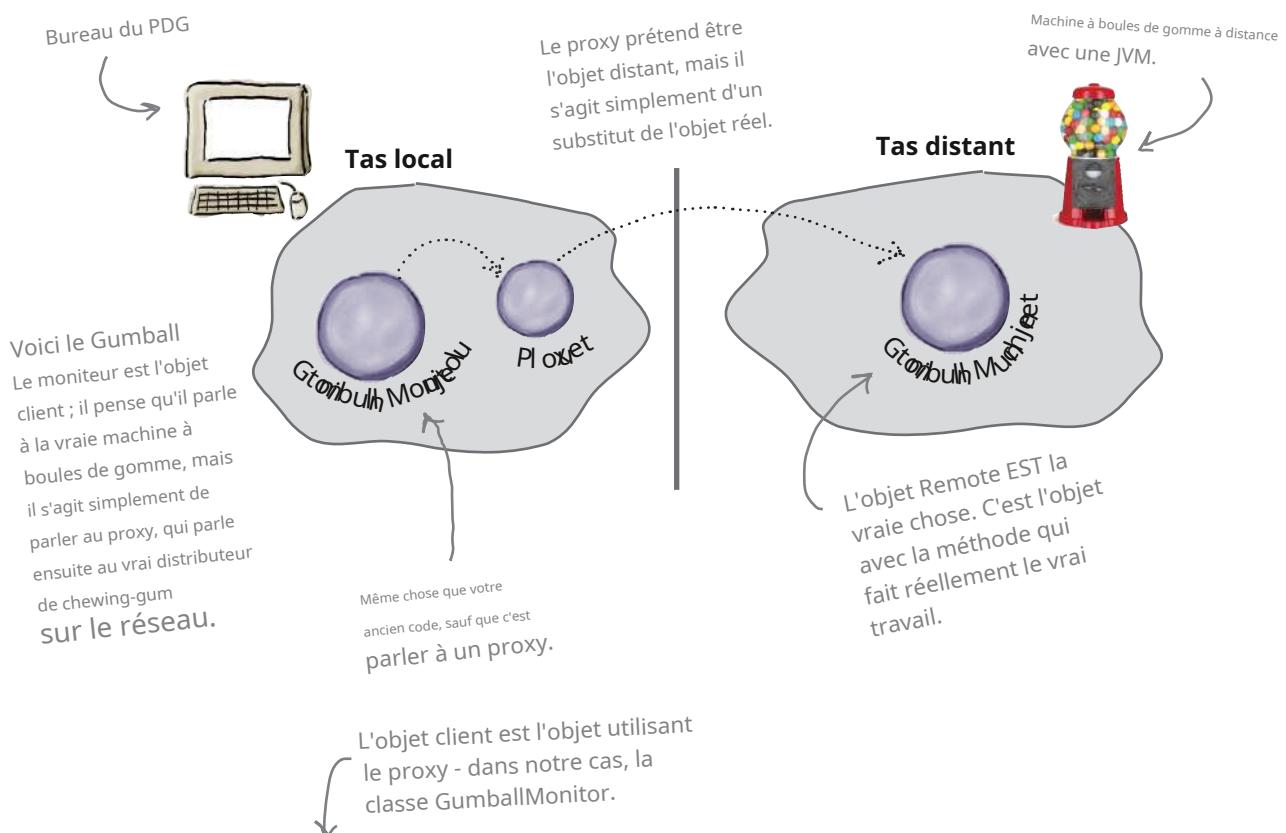
Joe :Ouais, c'est à peu près l'histoire.

Franc:Cela semble être quelque chose de plus facile à dire qu'à faire.

Joe :Peut-être, mais je ne pense pas que ce sera si mal. Nous devons nous assurer que le distributeur de chewing-gum peut agir comme un service et accepter les requêtes sur le réseau ; nous devons également donner à notre moniteur un moyen d'obtenir une référence à un objet proxy, mais nous avons déjà d'excellents outils intégrés à Java pour nous aider. Parlons d'abord un peu plus des proxys distants...

Le rôle du « proxy distant »

Un proxy distant agit comme un *représentant local d'un objet distant*. Qu'est-ce qu'un « objet distant » ? Il s'agit d'un objet qui réside dans le tas d'une autre machine virtuelle Java (ou plus généralement, d'un objet distant qui s'exécute dans un espace d'adressage différent). Qu'est-ce qu'un « représentant local » ? Il s'agit d'un objet sur lequel vous pouvez appeler des méthodes locales et les faire transférer vers l'objet distant.



Votre objet client agit comme s'il effectuait des appels de méthodes distantes. Mais en réalité, il appelle des méthodes sur un objet « proxy » heaplocal qui gère tous les détails de bas niveau de la communication réseau.



BRAIN POWER

Avant d'aller plus loin, réfléchissez à la manière dont vous concevriez un système permettant l'invocation de méthodes à distance (RMI). Comment faciliteriez-vous la tâche du développeur afin qu'il ait à écrire le moins de code possible ? Comment rendriez-vous l'invocation à distance transparente ?

BRAIN² POWER

Les appels à distance doivent-ils être totalement transparents ? Est-ce une bonne idée ? Quel pourrait être le problème avec cette approche ?

Ajout d'un proxy distant au code de surveillance de Gumball Machine

Sur le papier, notre plan semble bon, mais comment créer un proxy qui sait comment invoquer une méthode sur un objet qui réside dans une autre JVM ?

Hmmm. Eh bien, vous ne pouvez pas obtenir une référence à quelque chose sur un autre tas, n'est-ce pas ? En d'autres termes, vous ne pouvez pas dire :

Canard d = <objet dans un autre tas>

Quelle que soit la variable le d est réf. Le rencing doit être dans le même espace de tas que le code exécutant l'instruction. Alors, comment aborder cela ? Eh bien, c'est là qu'intervient l'invocation de méthode distante (RMI) de Java... RMI nous donne un moyen de trouver des objets dans une JVM distante et nous permet d'invoquer leurs méthodes.

C'est peut-être le bon moment pour réviser RMI avec votre référence Java préférée, ou vous pouvez suivre le [Détour RMI](#) en avant, et nous vous expliquerons les points forts de RMI avant d'ajouter le support proxy au code Gumball Machine.

Dans les deux cas, voici notre plan :

- 1 Tout d'abord, nous allons prendre le détour par RMI et explorer RMI. Même si vous connaissez RMI, vous voudrez peut-être le suivre et admirer le paysage.**
- 2 Ensuite, nous allons prendre notre machine Gumball et en faire un service distant qui fournit un ensemble d'appels de méthodes qui peuvent être invoqués à distance.**
- 3 Enfin, nous allons créer un proxy qui peut communiquer avec un distributeur de chewing-gum distant, toujours en utilisant RMI, et reconstituer le système de surveillance afin que le PDG puisse surveiller n'importe quel nombre de machines distantes.**

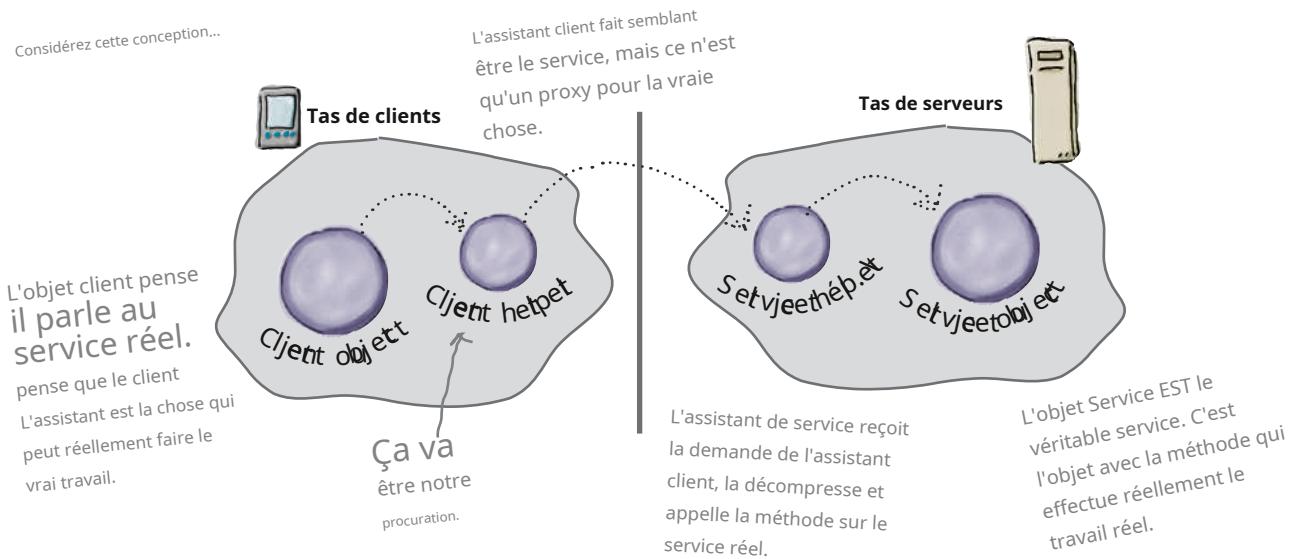


Un détour par le RMI

Si vous débutez avec RMI, faites le détour qui se déroule sur les pages suivantes. Sinon, vous pouvez simplement parcourir rapidement le détour pour réviser. Si vous souhaitez continuer, simplement pour comprendre l'essentiel du proxy distant, c'est également possible : vous pouvez ignorer le détour.



Méthodes à distance 101



Parcourir la conception

Supposons que nous souhaitons concevoir un système qui nous permette d'appeler un objet local qui transmet chaque requête à un objet distant. Comment le concevrons-nous ? Nous aurions besoin de quelques objets d'assistance qui effectueraient la communication pour nous. Les assistants permettent au client d'agir comme s'il appelait une méthode sur un objet local (ce qui est le cas). Le client appelle une méthode sur l'assistant client, comme si l'assistant client était le service réel. L'assistant client se charge ensuite de transmettre cette requête pour nous.

En d'autres termes, l'objet client pense qu'il appelle une méthode sur le service distant, car l'assistant client prétend être l'objet de service, c'est-à-dire prétend être l'objet avec la méthode que le client veut appeler.

Mais l'assistant client n'est pas réellement le service distant. Bien que l'assistant client se comporte comme tel (car il a la même méthode que celle annoncée par le service), l'assistant client n'a aucune des logiques de méthode attendues par le client. Au lieu de cela, l'assistant client contacte le serveur, transfère des informations sur l'appel de méthode (par exemple, le nom de la méthode, les arguments, etc.) et attend un retour du serveur.

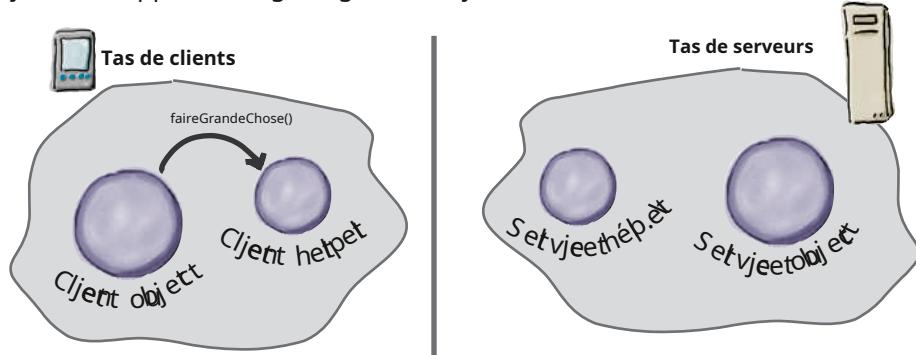
Côté serveur, l'assistant de service reçoit la requête de l'assistant client (via une connexion Socket), décomprime les informations sur l'appel, puis appelle la méthode réelle sur l'objet de service réel. Ainsi, pour l'objet de service, l'appel est local. Il provient de l'assistant de service, pas d'un client distant.

L'assistant de service récupère la valeur de retour du service, la compresse et la renvoie (via le flux de sortie d'un socket) à l'assistant client. L'assistant client décomprime les informations et renvoie la valeur à l'objet client.

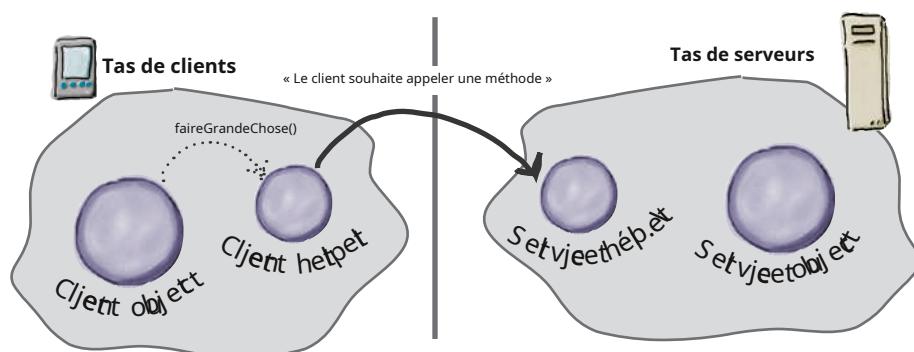
Passons en revue tout cela pour que ce soit plus clair...

Comment se déroule l'appel de méthode

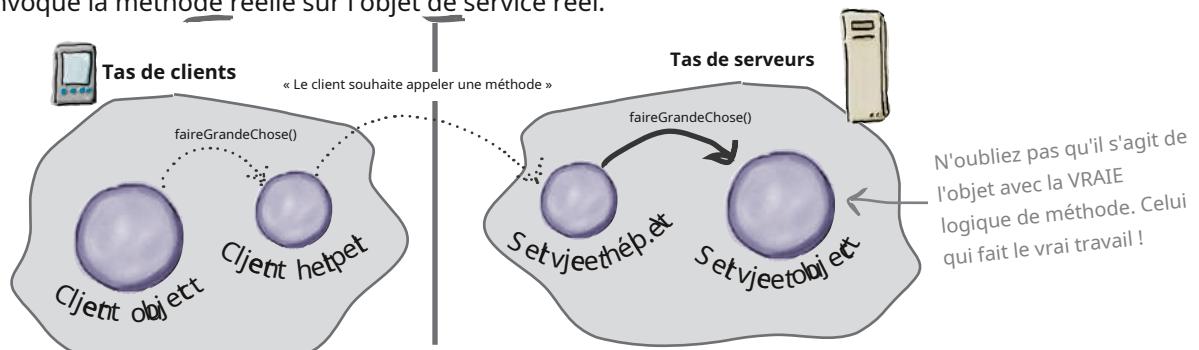
- ① l'objet Client appelle doBigThing() sur l'objet d'assistance client.



- ② l'assistant client regroupe les informations sur l'appel (arguments, nom de la méthode, etc.) et l'envoie sur le réseau à l'assistant de service.

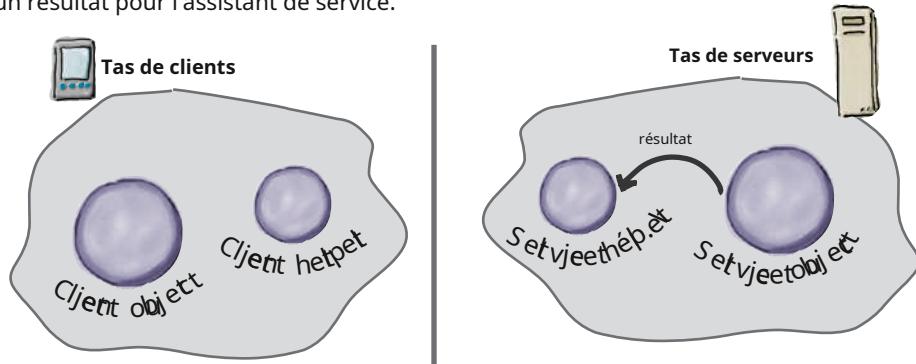


- ③ l'assistant de service décomprime les informations du client helper, découvre quelle méthode appeler (et sur quel objet) et invoque la méthode réelle sur l'objet de service réel.

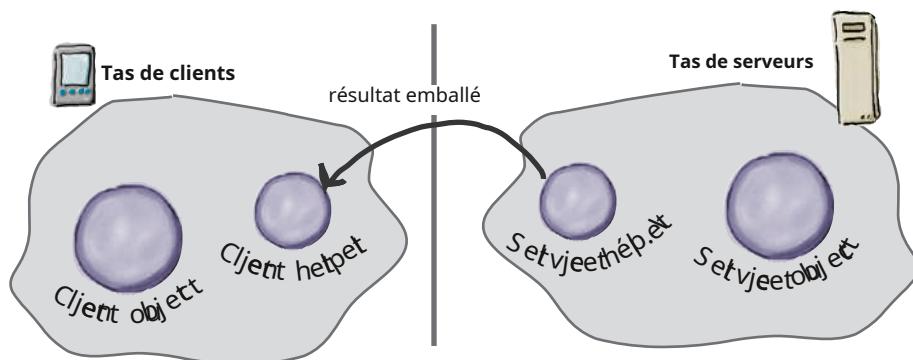




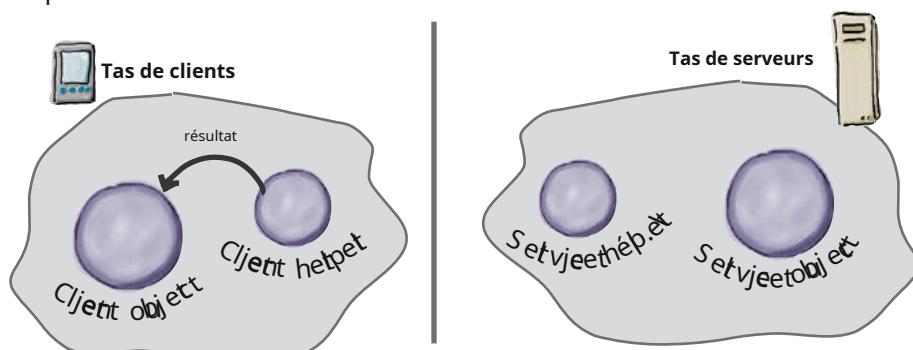
- ④ La méthode est invoquée sur l'objet de service, qui renvoie un résultat pour l'assistant de service.



- ⑤ L'assistant de service regroupe les informations renvoyées par le appel et renvoie-le sur le réseau à l'assistant client.



- ⑥ L'assistant client décomprime les valeurs renvoyées et renvoie vers l'objet client. Pour l'objet client, tout cela était transparent.



Java RMI, vue d'ensemble



OK, vous avez compris l'essentiel du fonctionnement des méthodes distantes ; il vous suffit maintenant de comprendre comment utiliser RMI.

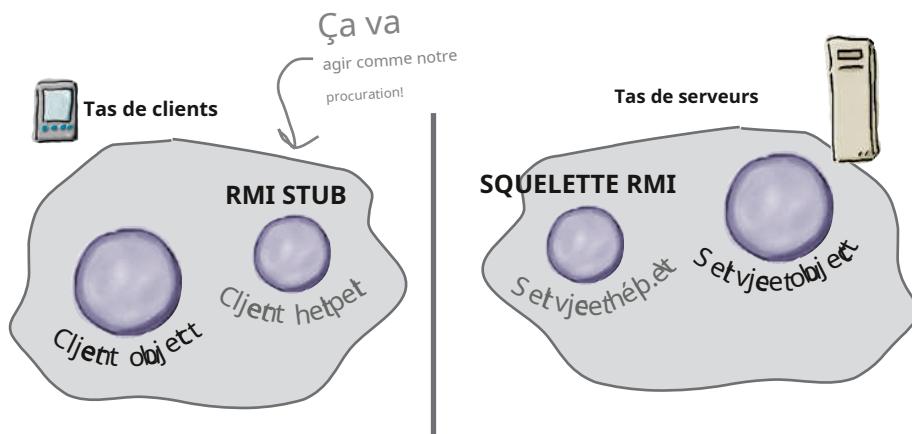
RMI vous permet de créer les objets d'assistance client et service, jusqu'à la création d'un objet d'assistance client avec les mêmes méthodes que le service distant. L'avantage de RMI est que vous n'avez pas à écrire vous-même le code réseau ou d'E/S. Avec votre client, vous appelez des méthodes distantes (c'est-à-dire celles dont dispose le service réel) comme des appels de méthodes normaux sur des objets exécutés dans la JVM locale du client.

RMI fournit également toute l'infrastructure d'exécution nécessaire pour que tout fonctionne, y compris un service de recherche que le client peut utiliser pour rechercher et accéder aux objets distants.

Il existe une différence entre les appels RMI et les appels de méthode locaux (normaux). N'oubliez pas que même si pour le client, l'appel de méthode semble être local, l'assistant client envoie l'appel de méthode sur le réseau. Il y a donc un réseau et des E/S. Et que savons-nous des réseaux et des méthodes d'E/S ?

Elles sont risquées ! Elles peuvent échouer ! Elles génèrent donc des exceptions à tout va. Le client doit donc reconnaître le risque. Nous verrons comment dans quelques pages.

Nomenclature RMI : dans RMI, l'assistant client est un « stub » et l'assistant service est un « squelette ».



Passons maintenant en revue toutes les étapes nécessaires pour transformer un objet en un service capable d'accepter des appels distants, ainsi que les étapes nécessaires pour permettre à un client de passer des appels distants.

Assurez-vous que votre ceinture de sécurité est bien attachée : il y a de nombreuses étapes à suivre, mais rien de trop inquiétant.



Un détour par le RMI

Création du service à distance

C'est un **aperçus** des cinq étapes de création du service distant, autrement dit les étapes nécessaires pour prendre un objet ordinaire et le surcharger afin qu'il puisse être appelé par un client distant. Nous ferons cela plus tard avec notre machine Gumball. Pour l'instant, décrivons les étapes, puis nous expliquerons chacune d'elles en détail.

Première étape :

Faire un**Interface à distance**

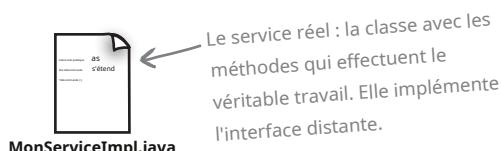
L'interface distante définit les méthodes qu'un client peut appeler à distance. C'est ce que le client utilisera comme type de classe pour votre service. Le stub et le service réel implémenteront ceci.



Deuxième étape :

Faire un**Mise en œuvre à distance**

Il s'agit de la classe qui fait le vrai travail. Elle contient l'implémentation réelle des méthodes distantes définies dans l'interface distante. C'est l'objet sur lequel le client souhaite appeler des méthodes (par exemple, GumballMachine).



Troisième étape :

Démarrer le**Registre RMI**(registerrm)

Le registerrm est comme les pages blanches d'un annuaire téléphonique. C'est là que le client se rend pour obtenir le proxy (l'objet stub/helper du client).

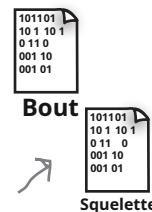


Exécutez ceci dans une fenêtre de terminal séparée.

Étape 4 :

Démarrer le**service à distance**

Vous devez mettre en place et exécuter l'objet de service. Votre classe d'implémentation de service instancie une instance du service et l'enregistre auprès du registre RMI. Son enregistrement rend le service disponible pour les clients.



Le Stub et le Skeleton sont générés dynamiquement pour vous en coulisses.

Première étape : créer une interface à distance



Un détour par le RMI

① Étendre java.rmi.Remote

Remote est une interface « marqueur », ce qui signifie qu'elle n'a pas de méthodes. Elle a cependant une signification particulière pour RMI, vous devez donc suivre cette règle. Notez que nous disons ici « étend ». Une interface est autorisée à étendre une autre interface.

interface publique MyRemote {

étend la télécommande {

Cela nous indique que l'interface va être utilisée pour prendre en charge les appels distants.

② Déclarer que toutes les méthodes lancent RemoteException

L'interface distante est celle que le client utilise comme type pour le service. En d'autres termes, le client invoque des méthodes sur quelque chose qui implémente l'interface distante. Ce quelque chose est le stub, bien sûr, et comme le stub fait du réseau et des E/S, toutes sortes de mauvaises choses peuvent se produire. Le client doit reconnaître les risques en gérant ou en déclarant les exceptions distantes. Si les méthodes d'une interface déclarent des exceptions, tout code appelant des méthodes sur une référence de ce type (le type d'interface) doit gérer ou déclarer les exceptions.

importer java.rmi.*;



L'interface distante est dans java.rmi.

interface publique MyRemote étend Remote {

public String sayHello() renvoie

Exception à distance;

}



Chaque méthode à distance l'appel est considéré «risqué». Déclarant RemoteException activée chaque méthode oblige le client à faire attention et à reconnaître que les choses pourraient ne pas fonctionner.

③ Assurez-vous que les arguments et les valeurs de retour sont primitifs ou sérialisables

Les arguments et les valeurs de retour d'une méthode distante doivent être soit primitifs, soit sérialisables. Pensez-y. Tout argument d'une méthode distante doit être empaqueté et expédié sur le réseau, et cela se fait via la sérialisation. La même chose s'applique aux valeurs de retour. Si vous utilisez des primitives, des chaînes et la majorité des types de l'API (y compris les tableaux et les collections), tout ira bien. Si vous transmettez vos propres types, assurez-vous simplement que vos classes implémentent Serializable.

publique Chaîne sayHello() lève RemoteException ;



Cette valeur de retour va être envoyée par câble du serveur au client, elle doit donc être sérialisable. C'est ainsi que les arguments et les valeurs de retour sont empaquetés et envoyés.

Découvrez votre Java préféré référence si vous besoin de rafraîchir votre mémoire sur Serializable.



Deuxième étape : réaliser une implémentation à distance

① Implémenter l'interface à distance

Votre service doit implémenter l'interface distante, celle avec les méthodes que votre client va appeler.

classe publique MyRemoteImpl étend UnicastRemoteObject

```
chaîne publique sayHello() {
    renvoie « Le serveur dit : « Hé » » ;
}
// plus de code dans la classe
}
```

implémente MyRemote {

Le compilateur s'assurera que vous avez implémenté toutes les méthodes de l'interface que vous implémentez. Dans ce cas, il n'y en a qu'une.

② Étendre UnicastRemoteObject

Pour fonctionner comme un objet de service distant, votre objet a besoin de certaines fonctionnalités liées au fait d'être « distant ». Le moyen le plus simple est d'étendre UnicastRemoteObject (à partir du package java.rmi.server) et de laisser cette classe (votre superclasse) faire le travail à votre place.

classe publique MyRemoteImpl étend UnicastRemoteObject implémente MyRemote {

privé statique final long serialVersionUID = 1L;

UnicastRemoteObject implémente Serializable, nous avons donc besoin du champ serialVersionUID.

③ Écrivez un constructeur sans argument qui déclare RemoteException

Votre nouvelle superclasse, UnicastRemoteObject, a un petit problème : son constructeur génère une exception RemoteException. La seule façon de résoudre ce problème est de déclarer un constructeur pour votre implémentation distante, juste pour avoir un endroit où déclarer RemoteException. N'oubliez pas que lorsqu'une classe est instanciée, son constructeur de superclasse est toujours appelé. Si votre constructeur de superclasse génère une exception, vous n'avez pas d'autre choix que de déclarer que votre constructeur génère également une exception.

public MyRemoteImpl() **génère RemoteException { }**

Vous n'avez rien à mettre dans le constructeur. Vous avez juste besoin d'un moyen de déclarer que le constructeur de votre superclasse génère une exception.

④ Enregistrer le service auprès du registre RMI

Maintenant que vous avez un service distant, vous devez le rendre disponible aux clients distants. Pour ce faire, vous l'instanciez et le placez dans le registre RMI (qui doit être en cours d'exécution, sinon cette ligne de code échoue). Lorsque vous enregistrez l'objet d'implémentation, le système RMI place en fait l'objet d'implémentation *bout* dans le registre, car c'est ce dont le client a réellement besoin. Enregistrez votre service en utilisant la méthode statique rebind() de la classe java.rmi.Naming.

essayer {

Mon service à distance = nouveau MyRemoteImpl();

Nommage.rebind("RemoteHello", service);

} catch(Exception ex) {...}

Donnez un nom à votre service (que les clients peuvent utiliser pour le rechercher dans le registre) et enregistrez-le dans le registre RMI. Lorsque vous liez l'objet de service, RMI échange le service contre le stub et place le stub dans le registre.

Troisième étape : exécuter rmiregistry



⑩ Ouvrez un terminal et démarrez le rmiregistry.

Assurez-vous de le démarrer à partir d'un répertoire qui a accès à vos classes. Le moyen le plus simple est de le démarrer à partir de votre cours annuaire.

```
Fenêtre d'édition de fichier Aide Hein ?  
Registre %rmi
```

Étape 4 : démarrer le service

① Ouvrez un autre terminal et démarrez votre service

Cela peut provenir d'une méthode main() de votre classe d'implémentation distante ou d'une classe de lancement distincte. Dans cet exemple simple, nous avons placé le code de démarrage dans la classe d'implémentation, dans une méthode main qui instancie l'objet et l'enregistre dans le registre RMI.

```
Fenêtre d'édition de fichier Aide Hein ?  
%java MyRemoteImpl
```

there are no
Dumb Questions

Q:

Pourquoi affichez-vous des stubs et des squelettes dans les diagrammes du code RMI ? Je pensais que nous nous en étions débarrassés il y a longtemps.

UN:

Vous avez raison ; pour le squelette, le runtime RMI peut envoyer les appels client directement au service distant à l'aide de la réflexion, et les stubs sont générés dynamiquement à l'aide de Dynamic Proxy (dont vous en apprendrez plus un peu plus loin dans le chapitre). Le stub de l'objet distant est une instance java.lang.reflect.Proxy (avec un gestionnaire d'invocation) qui est générée automatiquement pour gérer tous les détails de l'obtention des appels de méthode locale par le client à l'objet distant. Mais nous aimons montrer à la fois le stub et le squelette, car conceptuellement, cela vous aide à comprendre qu'il y a quelque chose sous les couvertures qui fait que cette communication entre le stub client et le service distant se produit.



Un détour par le RMI

Code complet pour le côté serveur

Jetons un œil à tout le code côté serveur :

L'interface à distance :

```
importer java.rmi.*;           RemoteException et l'interface Remote se
                                trouvent dans le package java.rmi.
```

```
interface publique MyRemote étend Remote {           Votre interface DOIT étendre java.rmi.Remote.
```

```
    public String sayHello() lève RemoteException ;           Toutes vos méthodes distantes doivent
}                                                 déclarer RemoteException.
```

Le service à distance (l'implémentation) :

```
importer java.rmi.*;           UnicastRemoteObject se trouve
                                dans le package java.rmi.server.
```

```
importer java.rmi.server.*;     L'extension de UnicastRemoteObject est le moyen
                                le plus simple de créer un objet distant.
```

```
classe publique MyRemoteImpl étend UnicastRemoteObject implémente MyRemote {           Vous DEVEZ mettre en œuvre
    privé statique final long serialVersionUID = 1L;           votre interface à distance !!
```

```
    chaîne publique sayHello() {           Bien entendu, vous devez implémenter
        renvoie « Le serveur dit : « Hé » » ;           toutes les méthodes d'interface. Mais
    }                                                 notez que vous n'avez PAS besoin de
                                                       déclarer RemoteException.
```

```
public MyRemoteImpl() lève RemoteException {}           Votre constructeur de superclasse (pour
                                                       UnicastRemoteObject) déclare une exception, VOUS
                                                       devez donc écrire un constructeur, car cela signifie
                                                       que votre constructeur appelle du code risqué (son
                                                       super constructeur).
```

```
public static void main (String[] args) {           Créez l'objet distant, puis « liez-le » au registre RMI à l'aide
    essayer {                                         de la fonction statique Naming.rebind(). Le nom sous lequel
        Mon service à distance = nouveau           vous l'enregistrez est le nom que les clients utiliseront pour
        MyRemoteImpl(); Naming.rebind("RemoteHello",     le rechercher dans le registre RMI.
        service); } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```



Un détour par le RMI

Et c'est là qu'intervient le registre RMI.

Et vous avez raison, le client doit obtenir l'objet stub (notre proxy), car c'est sur cet objet que le client appellera les méthodes. Pour ce faire, le client effectue une « recherche », comme s'il consultait les pages blanches d'un annuaire téléphonique, et dit essentiellement : « Voici un nom, et je voudrais le stub qui va avec ce nom. »

Jetons un œil au code dont nous avons besoin pour rechercher et récupérer un objet stub.

Voici comment cela fonctionne
sur la page suivante.



Le code de près

Le client utilise toujours l'interface distante comme type de service. En fait, le client n'a jamais besoin de connaître le nom de classe réel de votre service distant.

Service MyRemote =

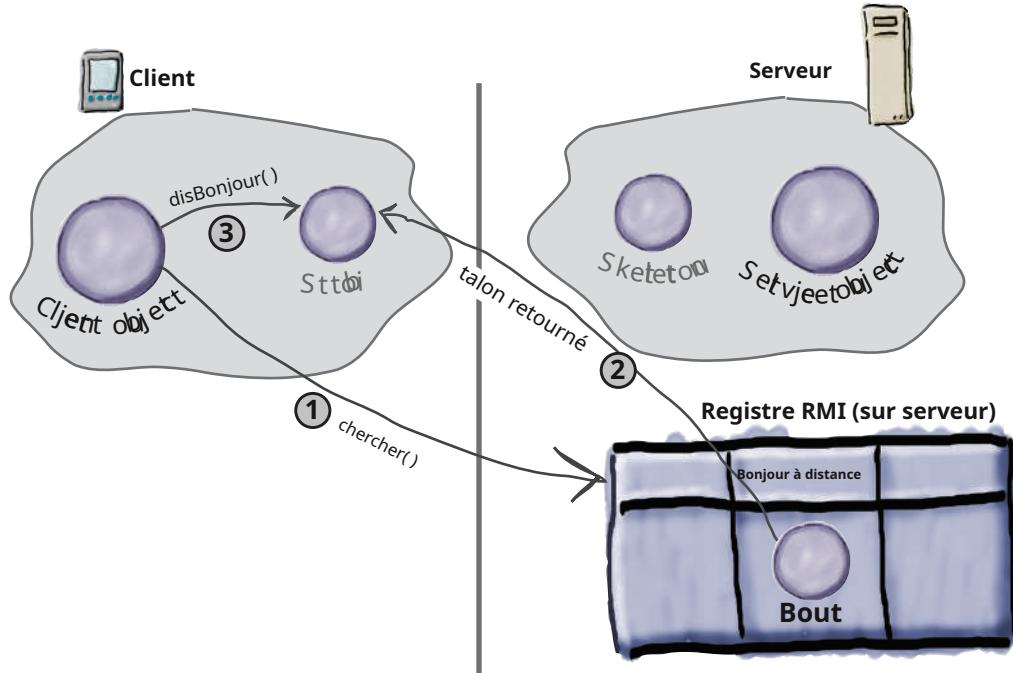
(MyRemote) Naming.lookup("rmi://127.0.0.1/RemoteHello");

Vous devez le convertir vers l'interface, car la méthode de recherche renvoie le type Object.

lookup() est une méthode statique de la classe Naming.

Il doit s'agir du nom sous lequel le service a été enregistré.

Le nom de l'hôte ou l'adresse IP où le service est en cours d'exécution.
(127.0.0.1 est localhost.)



Comment ça marche...

- ① Le client effectue une recherche dans le registre RMI
`Nommage.lookup("rmi://127.0.0.1/RemoteHello");`
- ② Le registre RMI renvoie l'objet stub
(comme valeur de retour de la méthode de recherche) et RMI déserialise automatiquement le stub.
- ③ Le client invoque une méthode sur le stub, comme si le stub ÉTAIT le vrai service

Code complet pour le côté client

Jetons un œil à tout le code côté client :

```

importer java.rmi.*;           ← La classe Naming (pour effectuer la recherche
                                rmiregistry) se trouve dans le package java.rmi.

classe publique MyRemoteClient {
    public static void main (String[] args) {
        nouveau MyRemoteClient().go();
    }

    public void go() {
        essayer {
            Service MyRemote = (MyRemote) Naming.lookup("rmi://127.0.0.1/RemoteHello");
            ↑
            ↑ Vous avez besoin de l'IP
            ↑ adresse ou nom d'hôte...
            ↑ ... et le nom utilisé pour
            ↑ lier/relier le service.

            Chaîne s = service.sayHello();
            ↑
            ↑ Cela ressemble à un appel de méthode
            ↑ classique ! (Sauf qu'il doit reconnaître
            ↑ l'exception RemoteException.)
            Système.out.println(s);
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}

```



Watch it!

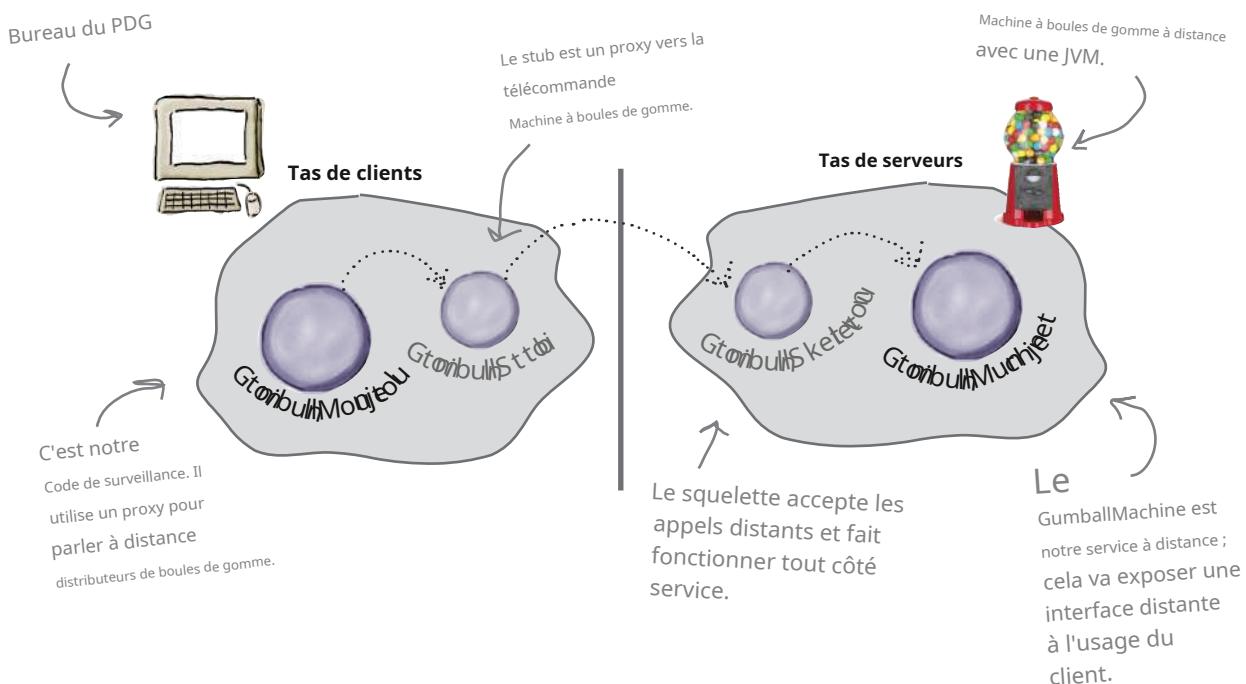
Les choses que les programmeurs font mal avec RMI sont :

1. Oubliez de démarrer rmiregistry avant de démarrer le service distant (lorsque le service est enregistré à l'aide de Naming, rebind(), rmiregistry doit être en cours d'exécution !)
2. Oubliez de rendre les arguments et les types de retour sérialisables (vous ne le saurez pas avant l'exécution ; ce n'est pas quelque chose que le compilateur détectera).



Retour à notre proxy distant GumballMachine

Bon, maintenant que vous avez les bases de RMI, vous disposez des outils dont vous avez besoin pour implémenter le proxy distant de la machine à boules de gomme. Voyons comment la machine à boules de gomme s'intègre dans ce cadre :



Arrêtez-vous et réfléchissez à la manière dont nous allons adapter le code du distributeur de chewing-gum pour qu'il fonctionne avec un proxy distant. N'hésitez pas à prendre quelques notes ici sur ce qui doit changer et ce qui sera différent de la version précédente.

Préparation de la machine à boules de gomme pour qu'elle

devienne un service à distance

La première étape de la conversion de notre code pour utiliser le proxy distant consiste à permettre à GumballMachine de traiter les requêtes distantes des clients. En d'autres termes, nous allons en faire un service. Pour ce faire, nous devons :

1. Créez une interface distante pour la GumballMachine. Cela fournira un ensemble de méthodes qui peuvent être appelées à distance.
2. Assurez-vous que tous les types de retour dans l'interface sont sérialisables.
3. Implémentez l'interface dans une classe concrète.

Nous allons commencer par l'interface distante :

```
N'oubliez pas d'importer java.rmi.*  
importer java.rmi.*;  
  
interface publique GumballMachineRemote étend Remote {  
    public int getCount() renvoie RemoteException; public String  
    getLocation() renvoie RemoteException; public State getState()  
    renvoie RemoteException;  
}
```

Tous les types de retour doivent être primitifs ou sérialisables...

Voici les méthodes que nous allons prendre en charge.
Chacune d'entre elles génère une RemoteException.

Nous avons un type de retour qui n'est pas sérialisable : la classe State. Résolvons ce problème...

```
importer java.io.*;
```

Serializable est dans le package java.io.

```
État de l'interface publique étend Serializable {  
    public void insertQuarter(); public  
    void ejectQuarter(); public void  
    turnCrank(); public void dispense();  
}
```

Ensuite, nous étendons simplement l'interface Serializable (qui ne contient aucune méthode). Et maintenant, l'état de toutes les sous-classes peut être transféré sur le réseau.

En fait, nous n'en avons pas encore fini avec Serializable ; nous avons un problème avec State. Comme vous vous en souvenez peut-être, chaque objet State conserve une référence à une machine à chewing-gum afin de pouvoir appeler les méthodes de la machine à chewing-gum et modifier son état. Nous ne voulons pas que la machine à chewing-gum soit entièrement sérialisée et transférée avec l'objet State. Il existe un moyen simple de résoudre ce problème :

```
la classe publique NoQuarterState implémente State {
    privé statique final long serialVersionUID = 2L;
    transitoire GumballMachine gumballMachine; // toutes les autres méthodes ici
}
```

Dans chaque implémentation de State, nous ajoutons le serialVersionUID et le mot-clé transient à la variable d'instance GumballMachine. Le mot-clé transient indique à la JVM de ne pas sérialiser ce champ. Notez que cela peut être légèrement dangereux si vous essayez d'accéder à ce champ une fois que l'objet a été sérialisé et transféré.

Nous avons déjà implémenté notre GumballMachine, mais nous devons nous assurer qu'elle peut agir comme un service et gérer les requêtes provenant du réseau. Pour ce faire, nous devons nous assurer que la GumballMachine fait tout ce qu'elle doit faire pour implémenter l'interface GumballMachineRemote.

Comme vous l'avez déjà vu dans le détour RMI, c'est assez simple ; il suffit d'ajouter quelques éléments...

```
Tout d'abord, nous devons importer les packages RMI.  

importer java.rmi.*;  

importer java.rmi.server.*;  

classe publique GumballMachine étend UnicastRemoteObject implémente GumballMachineRemote {  

    privé statique final long serialVersionUID = 2L;  

    // autres variables d'instance ici  

    public GumballMachine(Chaîne emplacement, int nombreGumballs)  

        // code ici  

    }  

    public int getCount() {  

        nombre de retours;  

    }  

    État public getState() {  

        état de retour;  

    }  

    chaîne publique getLocation() {  

        lieu de retour;  

    }  

    // autres méthodes ici
}
```

GumballMachine est va sous-classer l'UnicastRemoteObject ; cela lui donne la possibilité d'agir comme un service distant.

GumballMachine doit également implémenter l'interface distante...

génère une RemoteException {

... et le constructeur doit lancer une exception à distance, car la superclasse le fait.

C'est tout ! Rien aucun changement ici !

registre le service de boules de gomme

Inscription au registre RMI...

Cela termine le service de distributeur de chewing-gum. Il ne nous reste plus qu'à le démarrer pour qu'il puisse recevoir des demandes. Tout d'abord, nous devons nous assurer de l'enregistrer auprès du registre RMI afin que les clients puissent le localiser.

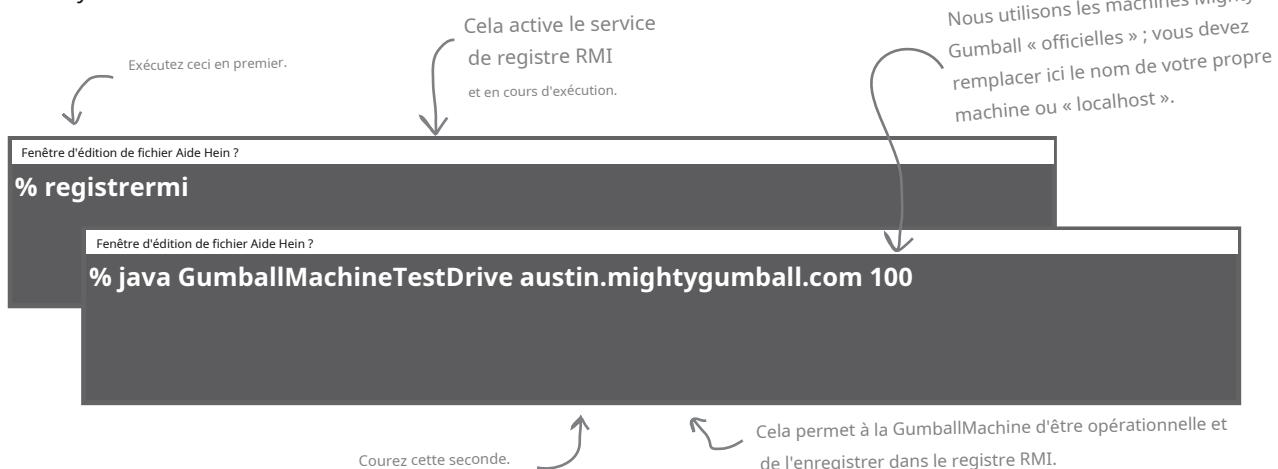
Nous allons ajouter un petit code au test drive qui s'occupera de cela pour nous :

```
classe publique GumballMachineTestDrive {  
  
    public static void main(String[] args) {  
        GumballMachineRemote gumballMachine = null; int  
        count;  
  
        si (args.length < 2) {  
            System.out.println("GumballMachine <nom> <inventaire>");  
            System.exit(1);  
        }  
  
        essayer {  
            compte = Integer.parseInt(args[1]);  
  
            gumballMachine = nouvelle GumballMachine(args[0], nombre);  
            Nommage.rebind("//" + args[0] + "/gumballmachine", gumballMachine);  
        } attraper (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Nous devons d'abord ajouter un bloc try/catch autour de l'instantiation de gumball car notre constructeur peut désormais générer des exceptions.

Nous ajoutons également l'appel à Naming.rebind, qui publie le stub GumballMachine sous le nom gumballmachine.

Allons-y et mettons-le en marche...



Passons maintenant au client GumballMonitor...

Vous vous souvenez de GumballMonitor ? Nous voulions le réutiliser sans avoir à le réécrire pour qu'il fonctionne sur un réseau. C'est en quelque sorte ce que nous allons faire, mais nous devons apporter quelques modifications.

```
importer java.rmi.*;           ← Nous devons importer le package RMI car nous
                                utilisons la classe RemoteException ci-dessous...

classe publique GumballMonitor {
    Machine à boules de gomme à distance machine;           ← Nous allons maintenant nous appuyer sur
                                                                l'interface distante plutôt que sur la classe
                                                                concrète GumballMachine.

    GumballMonitor public( Machine à boules de gomme à distance machine) {
        cette.machine = machine;
    }

    rapport public void() {
        essayer {
            System.out.println("Distributeur de chewing-gum : " + machine.getLocation());
            System.out.println("Inventaire actuel : " + machine.getCount() + " chewing-gums");
            System.out.println("État actuel : " + machine.getState());
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

← Nous devons également détecter toutes les exceptions distantes qui pourraient se produire lorsque nous essayons d'invoquer des méthodes qui se produisent finalement sur le réseau.



Rédaction du test de conduite du Monitor

Nous avons maintenant tous les éléments dont nous avons besoin. Il nous reste juste à écrire du code pour que le PDG puisse surveiller un tas de distributeurs de chewing-gum :

Voici le test du moniteur. Le PDG va l'exécuter !

```
importer java.rmi.*;  
  
classe publique GumballMonitorTestDrive {  
  
    public static void main(String[] args) {  
        Emplacement de la chaîne [] = {"rmi://santafe.mightygumball.com/gumballmachine",  
                                         "rmi://boulder.mightygumball.com/gumballmachine", "rmi://  
                                         austin.mightygumball.com/gumballmachine"};  
  
        GumballMonitor[] moniteur = nouveau GumballMonitor[emplacement.longueur];  
  
        pour (int i=0; i < location.length; i++) {  
            essayer {  
                Machine à boules de gommeMachine à distance =  
                    (GumballMachineRemote) Naming.lookup(location[i]);  
                monitor[i] = new GumballMonitor(machine);  
                System.out.println(monitor[i]);  
            } attraper (Exception e) {  
                e.printStackTrace();  
            }  
        }  
  
        pour (int i=0; i < longueur.du.moniteur; i++) {  
            surveiller[i].rapport();  
        }  
    }  
}
```

Voici tous les emplacements que nous allons surveiller.

Nous créons un tableau d'emplacements, un pour chaque machine.

Nous créons également une gamme de moniteurs.

Nous devons maintenant obtenir un proxy pour chaque machine distante.

Ensuite, nous parcourons chaque machine et imprimons son rapport.



Le code de près

```

Cela renvoie un proxy à la machine Gumball
distant (ou génère une exception si aucune ne
peut être localisée).

essayer {
    Machine à boules de gommeMachine à distance =
        (GumballMachineRemote) Nommage.lookup(location[i]);
}

moniteur[i] = nouveau GumballMonitor(machine);

} attraper (Exception e) {
    e.printStackTrace();
}

```

N'oubliez pas que Naming.lookup() est une méthode statique du package RMI qui prend un emplacement et un nom de service et le recherche dans le rmiregistry à cet endroit.

Une fois que nous obtenons un proxy vers la machine distante, nous créons un nouveau GumballMonitor et lui transmettons la machine à surveiller.

Une autre démo pour le PDG de Mighty Gumball...

Ok, il est temps de rassembler tout ce travail et de faire une autre démonstration. Commençons par nous assurer que quelques distributeurs de chewing-gum exécutent le nouveau code :

Sur chaque machine, exécutez rmiregistry en arrière-plan ou à partir d'une fenêtre de terminal distincte...

... puis exécutez la machine à chewing-gum, en lui donnant un emplacement et un nombre initial de boules de chewing-gum.

```

Fenêtre d'édition de fichier Aide Hein ?
% registre rmi &
% java GumballMachineTestDrive santafe.mightygumball.com 100

Fichier Ed ça Fenêtre Aide Hein ?
% registre rmi &
% java GumballMachineTestDrive boulder.mightygumball.com 100

Fenêtre d'édition de fichier Aide Hein ?
% registre rmi &
% java GumballMachineTestDrive austin.mightygumball.com 250

```

Machine populaire ! ↗

Et maintenant, mettons le moniteur entre les mains du PDG.
Espérons que cette fois, il l'aimera :

Fenêtre d'aide pour l'édition de fichiers GumballsAndBeyond

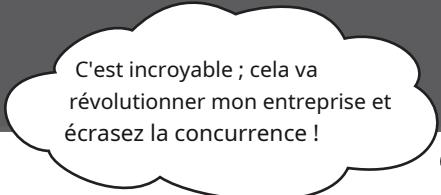
```
% java GumballMonitorTestDrive Distributeur de boules
de gomme : santafe.mightygumball.com Stock actuel :
99 boules de gomme
État actuel : en attente du trimestre

Distributeur de boules de gomme : boulder.mightygumball.com
Stock actuel : 44 boules de gomme
État actuel : en attente de tour de manivelle

Distributeur de boules de gomme : austin.mightygumball.com
Stock actuel : 187 boules de gomme
État actuel : en attente du quart %
```

Le moniteur itère
sur chaque télécommande
machine et appels
c'est getLocation(),
getCount(), et
méthodes getState().

C'est incroyable ; cela va
révolutionner mon entreprise et
écrasez la concurrence !



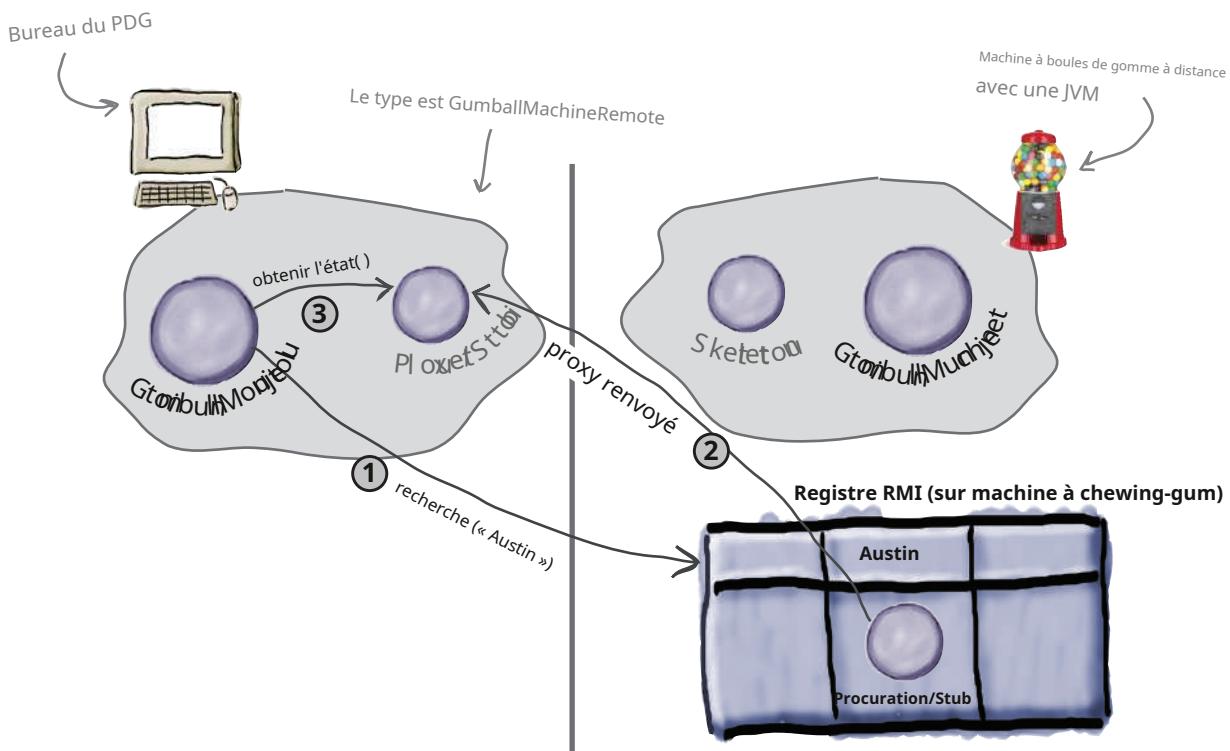
En invoquant des méthodes sur le proxy, nous effectuons un appel distant via le réseau et obtenons en retour une chaîne, un entier et un objet État. Étant donné que nous utilisons un proxy, Gumball Monitor ne sait pas, ou ne se soucie pas, que les appels sont distants (à part le fait de devoir se soucier des exceptions distantes).



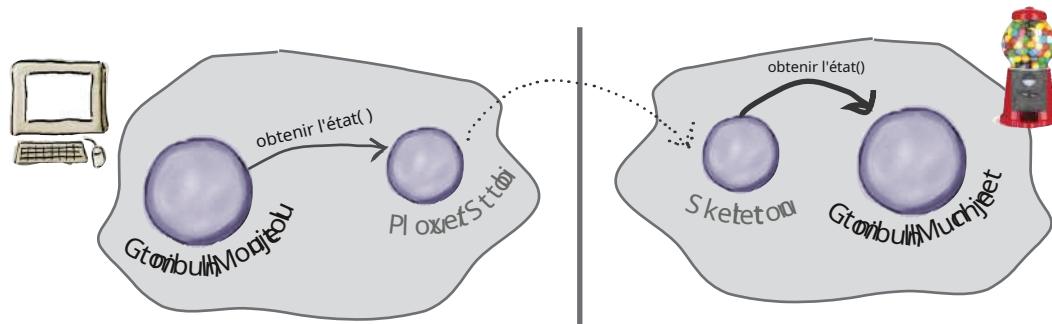
Derrière les scènes



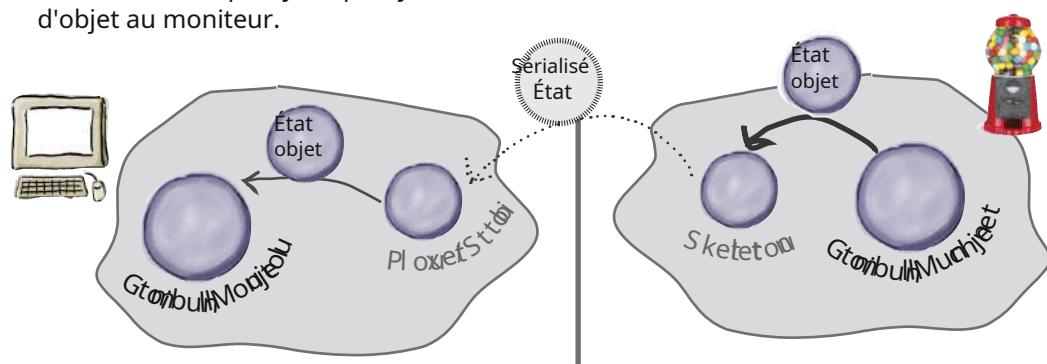
- Le PDG exécute le moniteur, qui récupère d'abord les proxys des machines à boules de gomme distantes, puis appelle getState() sur chacune d'elles (avec getCount() et getLocation()).



- ② getState() est appelé sur le proxy, qui transmet l'appel au service distant. Le squelette reçoit la requête puis la transmet à la GumballMachine.



- ③ GumballMachine renvoie l'état au squelette, qui le sérialise et le retransfère via le réseau au proxy. Le proxy le déserialise et le renvoie sous forme d'objet au moniteur.



Le moniteur n'a pas changé du tout, sauf qu'il sait qu'il peut rencontrer des exceptions distantes. Il utilise également l'interface GumballMachineRemote plutôt qu'une implémentation concrète.

De même, GumballMachine implémente une autre interface et peut lancer une exception distante dans son constructeur, mais à part cela, le code n'a pas changé.

Nous disposons également d'un petit bout de code pour enregistrer et localiser les stubs à l'aide du registre RMI. Mais quoi qu'il en soit, si nous écrivions quelque chose pour fonctionner sur Internet, nous aurions besoin d'une sorte de service de localisation.

Le modèle de proxy défini

Nous avons déjà parcouru de nombreuses pages dans ce chapitre. Comme vous pouvez le constater, l'explication du proxy distant est assez complexe. Malgré cela, vous verrez que la définition et le diagramme de classe du modèle de proxy sont en fait assez simples. Notez que le proxy distant est une implémentation du modèle de proxy général. Il existe en fait de nombreuses variantes du modèle, et nous en parlerons plus tard. Pour l'instant, découvrons les détails du modèle général.

Voici la définition du modèle de proxy :

Le modèle de proxyfournit un substitut ou un espace réservé à un autre objet pour contrôler l'accès à celui-ci.

Nous avons vu comment le modèle Proxy fournit un substitut ou un espace réservé pour un autre objet. Nous avons également décrit le proxy comme un « représentant » d'un autre objet.

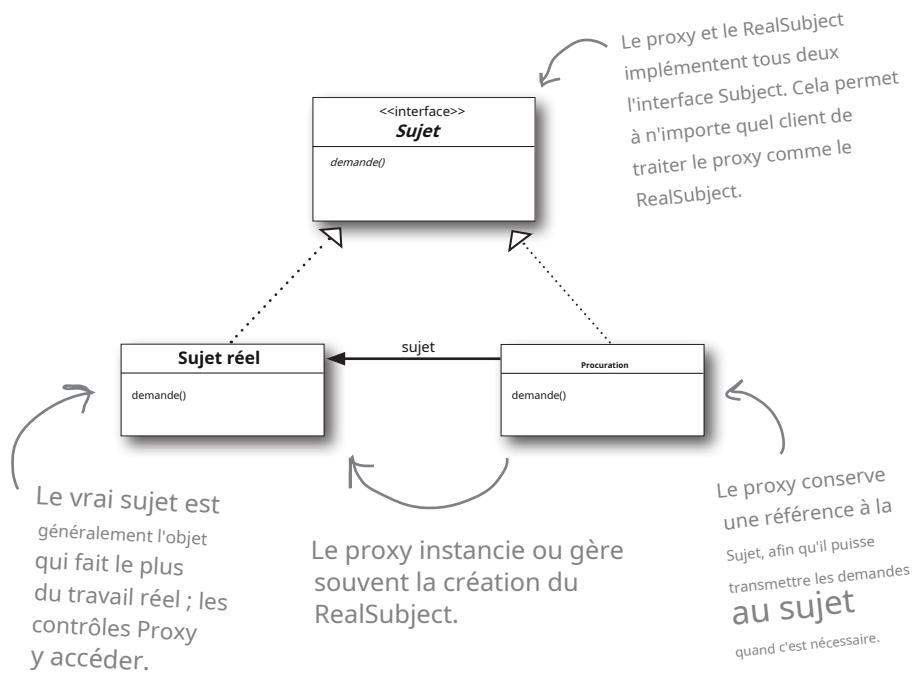
Mais qu'en est-il d'un proxy contrôlant l'accès ? Cela semble un peu étrange. Ne vous inquiétez pas. Dans le cas du distributeur de chewing-gum, pensez simplement au proxy contrôlant l'accès à l'objet distant. Le proxy avait besoin de contrôler l'accès car notre client, le moniteur, ne savait pas comment communiquer avec un objet distant. Donc, dans un certain sens, le proxy distant contrôlait l'accès afin de pouvoir gérer les détails du réseau pour nous. Comme nous venons de le voir, il existe de nombreuses variantes du modèle de proxy, et les variations tournent généralement autour de la façon dont le proxy « contrôle l'accès ». Nous en parlerons plus en détail plus tard, mais pour l'instant, voici quelques façons dont les proxys contrôlent l'accès :

- ƒ Comme nous le savons, un proxy distant contrôle l'accès à un objet distant.
- ƒ Un proxy virtuel contrôle l'accès à une ressource dont la création est coûteuse.
- ƒ Un proxy de protection contrôle l'accès à une ressource en fonction des droits d'accès.

Maintenant que vous avez compris l'essentiel du modèle général, regardez le diagramme de classe...

Utiliser le proxy
 Modèle pour créer un
 objet représentatif
 qui contrôle l'accès
 à un autre objet,
 qui peuvent être éloignés,
 coûteux à créer ou
 nécessitant d'être sécurisés.

le modèle proxy défini



Passons en revue le diagramme...

Nous avons d'abord un sujet, qui fournit une interface pour le RealSubject et le Proxy. Comme il implémente la même interface que le RealSubject, le Proxy peut remplacer le RealSubject partout où il apparaît.

Le RealSubject est l'objet qui effectue le véritable travail. C'est l'objet que le Proxy représente et dont il contrôle l'accès.

Le proxy contient une référence au RealSubject. Dans certains cas, le proxy peut être responsable de la création et de la destruction du RealSubject. Les clients interagissent avec le RealSubject via le proxy. Étant donné que le proxy et le RealSubject implémentent la même interface (Subject), le proxy peut être substitué partout où le Subject peut être utilisé. Le proxy contrôle également l'accès au RealSubject ; ce contrôle peut être nécessaire si le Subject s'exécute sur une machine distante, si le Subject est coûteux à créer d'une manière ou d'une autre, ou si l'accès au sujet doit être protégé d'une manière ou d'une autre.

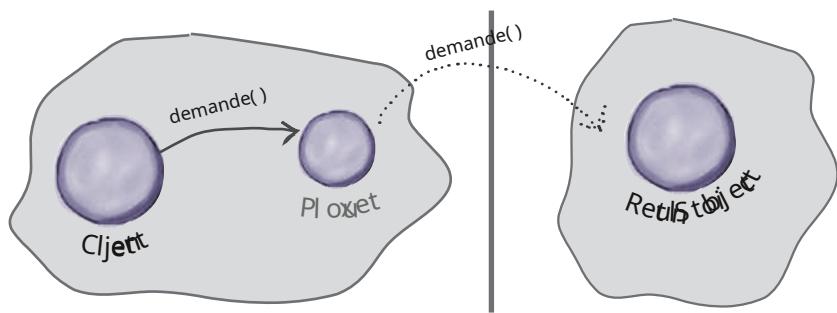
Maintenant que vous comprenez le modèle général, examinons d'autres façons d'utiliser un proxy au-delà du proxy distant...

Préparez-vous pour le proxy virtuel

Ok, jusqu'à présent, vous avez vu la définition du modèle proxy et vous avez examiné un exemple spécifique : le *Proxy à distance*. Nous allons maintenant examiner un autre type de proxy, le *Proxy virtuel*. Comme vous le découvrirez, le modèle de proxy peut se manifester sous de nombreuses formes, mais toutes les formes suivent à peu près la conception générale du proxy. Pourquoi autant de formes ? Parce que le modèle de proxy peut être appliqué à de nombreux cas d'utilisation différents. Examinons le proxy virtuel et comparons-le au proxy distant :

Proxy à distance

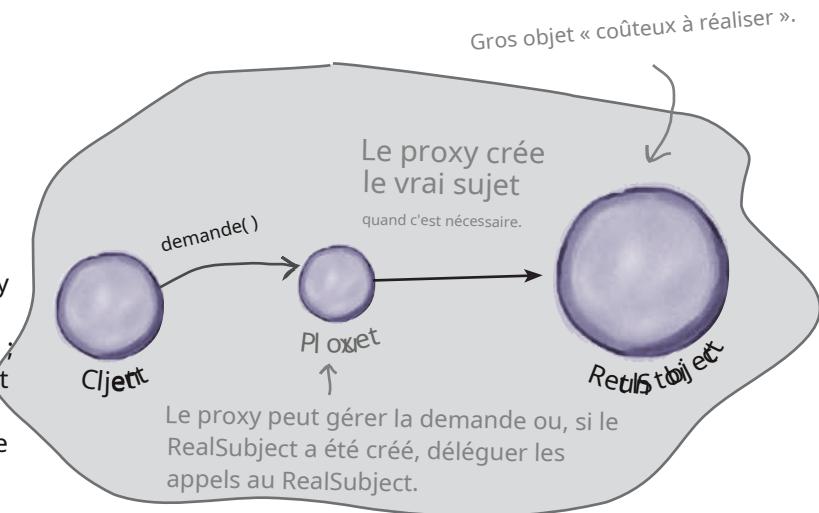
Avec le proxy distant, le proxy agit comme un représentant local pour un objet qui réside dans un JVM différente. Un appel de méthode sur le proxy entraîne le transfert de l'appel sur le réseau et son appel à distance, puis le retour du résultat au proxy, puis au client.



Nous connaissons assez bien ce diagramme maintenant...

Proxy virtuel

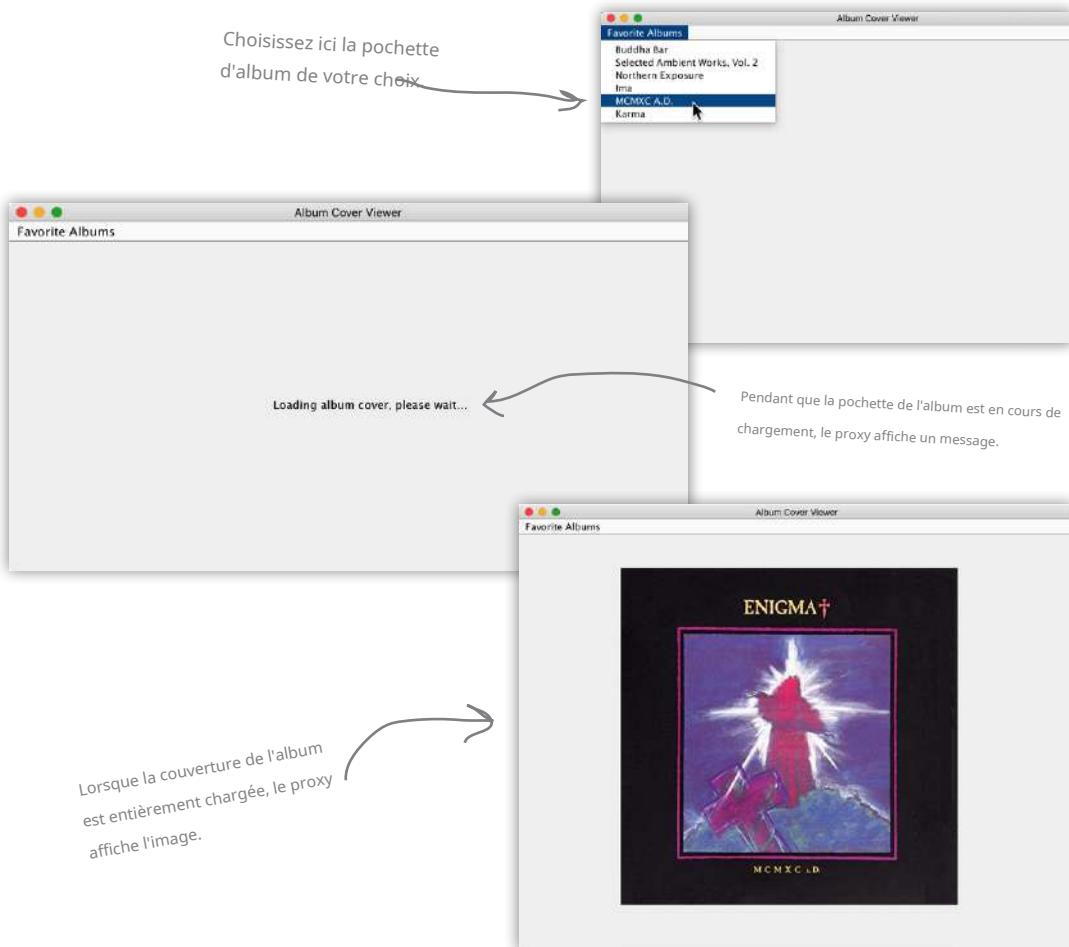
Le proxy virtuel agit comme un représentant d'un objet dont la création peut être coûteuse. Le proxy virtuel diffère souvent la création de l'objet jusqu'à ce qu'il soit nécessaire ; il agit également comme un substitut de l'objet avant et pendant sa création. Après cela, le proxy délègue les demandes directement au RealSubject.



Affichage des couvertures d'album

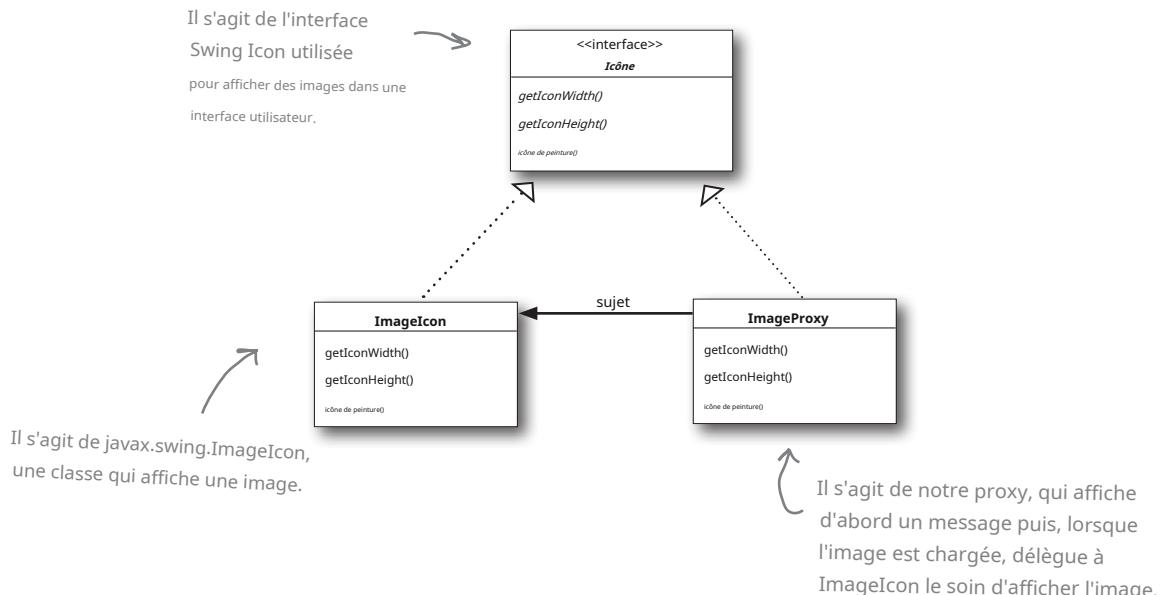
Supposons que vous souhaitiez écrire une application qui affiche vos pochettes d'album préférées. Vous pouvez créer un menu des titres d'album, puis récupérer les images à partir d'un service en ligne comme Amazon.com. Si vous utilisez Swing, vous pouvez créer une icône et lui demander de charger l'image à partir du réseau. Le seul problème est que, selon la charge du réseau et la bande passante de votre connexion, la récupération d'une pochette d'album peut prendre un peu de temps. Votre application doit donc afficher quelque chose pendant que vous attendez que l'image se charge. Nous ne voulons pas non plus bloquer toute l'application pendant qu'elle attend l'image. Une fois l'image chargée, le message devrait disparaître et vous devriez voir l'image.

Un moyen simple d'y parvenir est d'utiliser un proxy virtuel. Le proxy virtuel peut se substituer à l'icône, gérer le chargement en arrière-plan et, avant que l'image ne soit entièrement récupérée du réseau, afficher « Chargement de la couverture de l'album, veuillez patienter... ». Une fois l'image chargée, le proxy délégue l'affichage à l'icône.



Conception du proxy virtuel de la couverture de l'album

Avant d'écrire le code de l'Album Cover Viewer, examinons le diagramme de classes. Vous verrez qu'il ressemble à notre diagramme de classes Remote Proxy, mais ici le proxy est utilisé pour masquer un objet coûteux à créer (car nous devons récupérer les données de l'icône sur le réseau) par opposition à un objet qui se trouve en fait ailleurs sur le réseau.



Comment ImageProxy va fonctionner :

- 1** **ImageProxy crée d'abord une ImageIcon et commence à la charger à partir d'une URL réseau.**
- 2** **Pendant que les octets de l'image sont récupérés, ImageProxy affiche « Chargement de la couverture de l'album, veuillez patienter... ».**
- 3** **Lorsque l'image est entièrement chargée, ImageProxy délègue tous les appels de méthode à l'icône de l'image, y compris `paintIcon()`, `getIconWidth()` et `getIconHeight()`.**
- 4** **Si l'utilisateur demande une nouvelle image, nous créerons un nouveau proxy et recommencerons le processus.**

Écriture du proxy d'image

```

la classe ImageProxy implémente Icon {
    volatile ImageIcon imageIcon; URL
    finale imageURL;
    Récupération de threadThread;
    booléen récupération = false;

    public ImageProxy(URL url) { imageURL = url; } public int
    getIconWidth() {
        si (imageIcon != null) {
            retourner imageIcon.getIconWidth(); }
        sinon {
            retourner 800;
        }
    }
    public int getIconHeight() {
        si (imageIcon != null) {
            retourner imageIcon.getIconHeight(); }
        sinon {
            retourner 600;
        }
    }
    synchronisé void setImageIcon(ImageIcon imageIcon) {
        ceci.imageIcon = imageIcon;
    }

    public void paintIcon(composant final c, graphiques g, int x, int y) {
        si (imageIcon != null) {
            imageIcon.paintIcon(c, g, x, y); } else {

                g.drawString("Chargement de la couverture de l'album, veuillez patienter...", x+300,
                y+190); if (!retrieving) {
                    récupération = vrai;

                    retrievalThread = nouveau thread (nouveau runnable() {
                        public void run() {
                            essayer {
                                setImageIcon(new ImageIcon(imageURL, "Couverture d'album"));
                                c.repaint();
                            } attraper (Exception e) {
                                e.printStackTrace();
                            }
                        }
                    });
                    récupérationThread.start();
                }
            }
        }
    }
}

```

Le proxy d'image implémente l'icône interface.

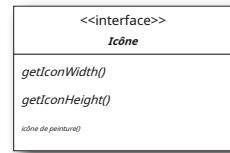
L'imageIcon est l'icône RÉELLE que nous souhaitons éventuellement afficher lorsqu'elle est chargée.

Nous passons l'URL de l'image au constructeur. C'est l'image que nous devons afficher une fois chargée !

Nous renvoyons une largeur et une hauteur par défaut jusqu'à ce que l'imageIcon soit chargée ; ensuite, nous la transmettons à l'imageIcon.

imageIcon est utilisé par deux threads différents, donc en plus de rendre la variable volatile (pour protéger les lectures), nous utilisons un setter synchronisé (pour protéger les écritures).

C'est là que les choses deviennent intéressantes. Ce code peint l'icône sur l'écran (en déléguant à imageIcon). Cependant, si nous n'avons pas d'imageIcon entièrement créée, nous en créons une. Regardons cela de plus près sur la page suivante...





Le code de près

```

    Cette méthode est appelée lorsqu'il est temps de peindre l'icône sur l'écran.
    ↘

public void paintIcon(composant final c, graphiques
    si (imageIcon != null) {
        imageIcon.paintIcon(c, g, x, y);
    } autre {
        g.drawString("Chargement de la couverture de l'album, veuillez patienter...", x+300,
        y+190); if (!retrieving) {
            récupération = vrai;
            retrievalThread = nouveau thread (nouveau runnable() {
                public void run() {
                    essayer {
                        setImageIcon(new ImageIcon(imageURL, "Couverture d'album"));
                        c.repaint();
                    } attraper (Exception e) {
                        e.printStackTrace();
                    }
                }
            });
            récupérationThread.start();
        }
    }
}

```

Si nous avons déjà une icône, nous lui disons de se peindre elle-même.

Sinon nous afficher le message de « chargement ».

C'est ici que nous chargeons l'image d'icône RÉELLE. Notez que le chargement de l'image avec ImageIcon est synchrone : le constructeur ImageIcon ne revient pas tant que l'image n'est pas chargée. Cela ne nous donne pas beaucoup de chance de faire des mises à jour d'écran et d'afficher notre message, nous allons donc le faire de manière asynchrone. Voir le « Code Way Up Close » sur la page suivante pour en savoir plus...



Le code de près

Si nous n'essayons pas déjà de récupérer l'image...

```
si (!récupération) {  
    récupération = vrai;  
  
    retrievalThread = nouveau thread (nouveau runnable) {  
        public void run() {  
            essayer {  
                setImageIcon(new ImageIcon(imageURL, "Couverture d'album"));  
                c.repaint();  
            } attraper (Exception e) {  
                e.printStackTrace();  
            }  
        };  
        récupérationThread.start();  
    }  
}
```

... alors il est temps de commencer à le récupérer (au cas où vous vous poseriez la question, un seul thread appelle `paint`, donc nous devrions être en sécurité ici).

Nous ne voulons pas bloquer toute l'interface utilisateur, nous allons donc utiliser un autre thread pour récupérer l'image.

Quand on a l'image, on dit à Swing qu'il faut la repeindre.

Dans notre fil, nous instancions le Objet icône. Son le constructeur ne le fera pas revenir jusqu'à ce que le l'image est chargée.

Ainsi, la prochaine fois que l'affichage est peint après linstanciation de `ImageIcon`, la méthode `paintIcon()` peindra l'image, pas le message de chargement.



Puzzle de conception

La classe ImageProxy semble avoir deux états contrôlés par des instructions conditionnelles. Pouvez-vous penser à un autre modèle qui pourrait nettoyer ce code ? Comment reconcevriez-vous ImageProxy ?

la classe ImageProxy implémente Icon {

// variables d'instance et constructeur ici

```
public int getIconWidth() {
    si (imageIcon != null) {
        retourner imageIcon.getIconWidth(); }
    sinon {
        retourner 800;
    }
}
```



```
public int getIconHeight() {
    si (imageIcon != null) {
        retourner imageIcon.getIconHeight(); }
    sinon {
        retourner 600;
    }
}
```



```
public void paintIcon(composant final c, graphiques g, int x, int y) {
    si (imageIcon != null) {
        imageIcon.paintIcon(c, g, x, y); } else {
            g.drawString("Chargement de la couverture de l'album, veuillez patienter...", x+300, y+190); //
            plus de code ici
    }
}
```



Test de la visionneuse de couvertures d'album



Prêt à cuire
Code

Ok, il est temps de tester ce nouveau proxy virtuel sophistiqué. Dans les coulisses, nous avons créé un nouveau ImageProxyTestDrive qui configure la fenêtre, crée un cadre, installe les menus et crée notre proxy. Nous ne passons pas en revue tout ce code dans les moindres détails ici, mais vous pouvez toujours récupérer le code source et y jeter un œil, ou le consulter à la fin du chapitre où nous listons tout le code source du proxy virtuel.

Voici une vue partielle du code de test :

```
classe publique ImageProxyTestDrive {  
    Composant imageComposant;  
    public static void main (String[] args) génère une exception {  
        ImageProxyTestDrive testDrive = nouveau ImageProxyTestDrive();  
    }  
}
```

```
public ImageProxyTestDrive() génère une exception {
```

```
    // configurer le cadre et les menus
```

Ici, nous créons un proxy d'image et le définissons sur une URL initiale. Chaque fois que vous choisissez une sélection dans le menu Album, vous obtenez un nouveau proxy d'image.

```
    Icône icône = nouveau ImageProxy(initialURL);  
    imageComponent = nouveau ImageComponent(icône);  
    frame.getContentPane().add(imageComponent);
```

Ensuite, nous enveloppons notre proxy dans un composant afin qu'il puisse être ajouté au cadre. Le composant s'occupera de la largeur, de la hauteur et des détails similaires du proxy.

```
}
```

```
}
```

Enfin, nous ajoutons le proxy au cadre afin qu'il puisse être affiché.

Exécutons maintenant le test routier :

```
Fenêtre d'édition de fichier Aide Quelques-uns des albums qui nous ont aidés à traverser ce livre  
% java ImageProxyTestDrive
```

L'exécution d'ImageProxyTestDrive devrait vous donner une fenêtre comme celle-ci.

Des choses à essayer...

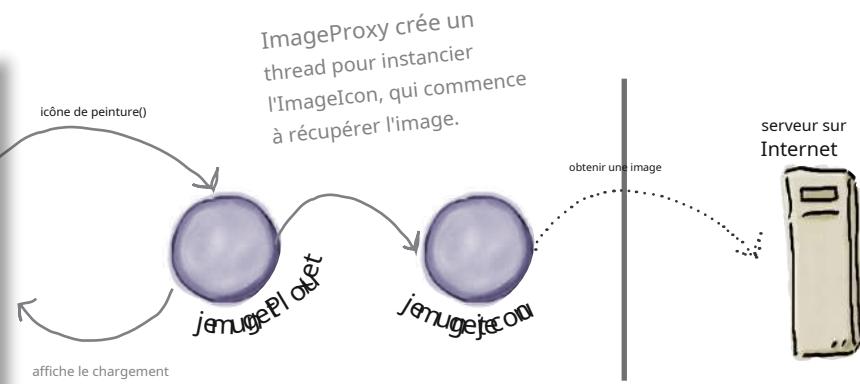
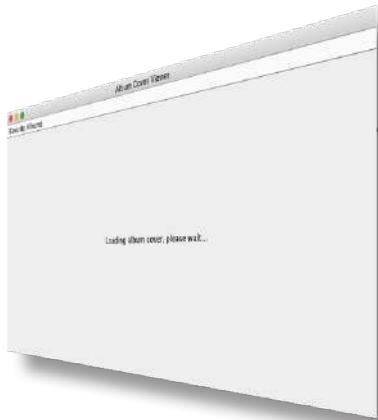
- ➊ Utilisez le menu pour charger différentes pochettes d'album ; regardez l'affichage proxy « chargement » jusqu'à ce que l'image soit arrivée.
- ➋ Redimensionnez la fenêtre lorsque le message « chargement » s'affiche. Notez que le proxy gère le chargement sans bloquer la fenêtre Swing.
- ➌ Ajoutez vos propres albums préférés à ImageProxyTestDrive.



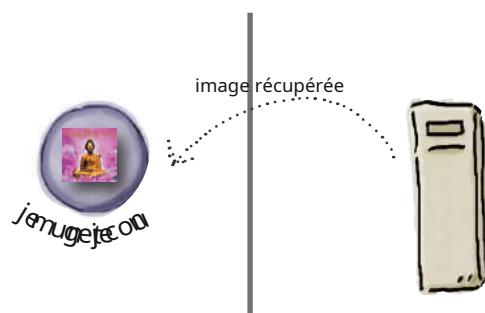
Qu'avons-nous fait ?

- 1 Nous avons créé une classe ImageProxy pour l'affichage.
La méthode paintIcon() est appelée et ImageProxy déclenche un thread pour récupérer l'image et créer l'ImageIcon.

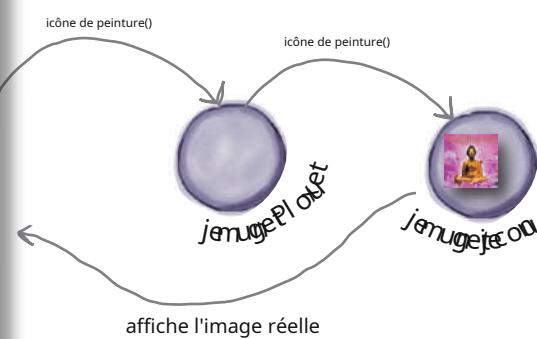
Derrière
les scènes



- 2 À un moment donné, l'image est renvoyée et l'ImageIcon entièrement instancié.



- 3 Une fois l'ImageIcon créée, la prochaine fois que paintIcon() est appelé, le proxy délègue à l'ImageIcon.



there are no Dumb Questions

Q: Le proxy distant et le proxy virtuel me semblent si différents ; sont-ils vraiment un seul modèle ?

UN: Vous trouverez de nombreuses variantes du modèle Proxy dans le monde réel. Elles ont toutes en commun d'intercepter une invocation de méthode que le client effectue sur le sujet. Ce niveau d'indirection nous permet de faire de nombreuses choses, notamment d'envoyer des requêtes à un sujet distant, de fournir un représentant pour un objet coûteux au fur et à mesure de sa création ou, comme vous le verrez, de fournir un certain niveau de protection qui peut déterminer quels clients doivent appeler quelles méthodes. Ce n'est que le début. Le modèle Proxy général peut être appliqué de nombreuses manières différentes, et nous aborderons certaines des autres manières à la fin du chapitre.

Q: *ImageProxy* me semble être juste un décorateur. Je veux dire, nous enveloppons essentiellement un objet avec un autre, puis nous déléguons les appels à *ImageIcon*. Qu'est-ce qui me manque ?

UN: Parfois, Proxy et Decorator se ressemblent beaucoup, mais leurs objectifs sont différents : un décorateur ajoute un comportement à une classe, tandis qu'un proxy contrôle l'accès à celle-ci. Vous pourriez vous demander : « Le message de chargement n'ajoute-t-il pas un comportement ? » D'une certaine manière, c'est le cas ; cependant, plus important encore, *ImageProxy* contrôle l'accès à une *ImageIcon*. Comment contrôle-t-il l'accès ? Eh bien, pensez-y de cette façon : le proxy découpe le client de l'*ImageIcon*. S'ils étaient couplés

le client doit attendre que chaque image soit récupérée avant de pouvoir peindre toute son interface. Le proxy contrôle l'accès à l'*ImageIcon* de sorte qu'avant qu'elle ne soit entièrement créée, le proxy fournit une autre représentation à l'écran. Une fois l'*ImageIcon* créée, le proxy autorise l'accès.

Q: Comment puis-je faire en sorte que les clients utilisent le proxy plutôt que le sujet réel ?

UN: Bonne question. Une technique courante consiste à fournir une fabrique qui instancie et renvoie le sujet. Étant donné que cela se produit dans une méthode de fabrique, nous pouvons ensuite envelopper le sujet avec un proxy avant de le renvoyer. Le client ne sait jamais ou ne se soucie jamais d'utiliser un proxy au lieu du vrai.

Q: J'ai remarqué que dans l'exemple *ImageProxy*, vous créez toujours une nouvelle *ImageIcon* pour obtenir l'image, même si l'image a déjà été récupérée. Pourriez-vous implémenter quelque chose de similaire à *ImageProxy* qui met en cache les récupérations passées ?

UN: Vous parlez d'une forme spécialisée de proxy virtuel appelée proxy de mise en cache. Un proxy de mise en cache conserve un cache d'objets créés précédemment et, lorsqu'une demande est effectuée, il renvoie un objet mis en cache, si possible.

Nous allons examiner cela et plusieurs autres variantes du modèle Proxy à la fin du chapitre.

Q: Je vois comment Decorator et Proxy sont liés, mais qu'en est-il de l'adaptateur ? Un adaptateur semble également très similaire.

UN: Le proxy et l'adaptateur se trouvent devant d'autres objets et leur transmettent les requêtes. N'oubliez pas que l'adaptateur modifie l'interface des objets qu'il adapte, tandis que le proxy implémente la même interface.

Il existe une autre similitude avec le proxy de protection. Un proxy de protection peut autoriser ou interdire l'accès d'un client à des méthodes particulières d'un objet en fonction du rôle du client. De cette façon, un proxy de protection peut uniquement fournir une interface partielle à un client, ce qui est assez similaire à certains adaptateurs. Nous allons examiner le proxy de protection dans quelques pages.

Fireside Chats



La conférence de ce soir :**Le proxy et le décorateur deviennent intentionnels.**

Procuration:

Bonjour, décorateur. Je suppose que vous êtes ici parce que les gens nous confondent parfois ?

Moi copier tonDes idées ? S'il vous plaît. Je contrôle l'accès aux objets. Vous les décorez simplement. Mon travail est tellement plus important que le vôtre, ce n'est même pas drôle.

Bon, peut-être que tu n'es pas entièrement frivole... mais je ne comprends toujours pas pourquoi tu penses que je copie toutes tes idées. Je m'intéresse à représenter mes sujets, pas à les décorer.

Je ne pense pas que tu comprennes, Décorateur. Je remplace mes Sujets ; je n'ajoute pas simplement un comportement. Les clients m'utilisent comme substitut d'un Sujet Réel, car je peux les protéger des accès indésirables, ou empêcher leurs interfaces graphiques de se bloquer pendant qu'ils attendent que de gros objets se chargent, ou cacher le fait que leurs Sujets s'exécutent sur des machines distantes. Je dirais que c'est une intention très différente de la tienne !

Décorateur:

Eh bien, je pense que la raison pour laquelle les gens nous confondent, c'est que vous vous promenez en prétendant être un modèle complètement différent, alors qu'en fait, vous n'êtes qu'un décorateur déguisé. Je ne pense vraiment pas que vous devriez copier toutes mes idées.

« Juste » décorer ? Vous pensez que décorer est une modèle frivole et sans importance ? Laisse-moi te dire mon pote, j'ajoute *comportement* C'est la chose la plus importante à propos des objets : ce qu'ils font !

Vous pouvez l'appeler « représentation », mais si cela ressemble à un canard et marche comme un canard... Je veux dire, regardez simplement votre proxy virtuel ; c'est juste une autre façon d'ajouter un comportement pour faire quelque chose pendant qu'un gros objet coûteux est en cours de chargement, et votre proxy distant est un moyen de communiquer avec des objets distants afin que vos clients n'aient pas à s'en soucier eux-mêmes. Tout est une question de comportement, comme je l'ai dit.

Appelez cela comme vous voulez. J'implémente la même interface que les objets que j'encapsule ; vous aussi.

Procuration:

Ok, passons en revue cette affirmation. Vous encapsulez un objet. Bien que nous disions parfois de manière informelle qu'un proxy encapsule son sujet, ce n'est pas vraiment un terme précis.

Pensez à un proxy distant... quel objet est-ce que j'encapsule ? L'objet que je représente et dont je contrôle l'accès se trouve sur une autre machine ! Voyons comment vous faites cela.

Bien sûr, d'accord, prenons un proxy virtuel... pensez à l'exemple du visualiseur d'album. Lorsque le client m'utilise pour la première fois comme proxy, le sujet n'existe même pas ! Alors, qu'est-ce que j'enveloppe là ?

Je ne savais pas que les décorateurs étaient si stupides ! Bien sûr, je crée parfois des objets. Comment penses-tu qu'un proxy virtuel obtienne son sujet ?! Ok, tu viens de souligner une grande différence entre nous : nous savons tous les deux que les décorateurs n'ajoutent que de la poudre aux yeux ; ils n'ont jamais le droit d'instancier quoi que ce soit.

Hé, après cette conversation, je suis convaincu que tu n'es qu'un simple mandataire !

Vous verrez très rarement un proxy envelopper un sujet plusieurs fois ; en fait, si vous enveloppez quelque chose 10 fois, vous feriez mieux de revenir en arrière et de réexaminer votre conception.

Décorateur:

Ah oui ? Pourquoi pas ?

D'accord, mais nous savons tous que les proxys distants sont un peu bizarres. Vous avez un deuxième exemple ? J'en doute.

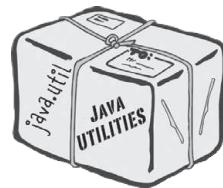
Euh, et la prochaine chose que vous direz, c'est que vous pourrez réellement créer des objets.

Ah oui ? Instanciez ceci !

Proxy stupide ? J'aimerais te voir envelopper récursivement un objet avec 10 décorateurs et garder la tête droite en même temps.

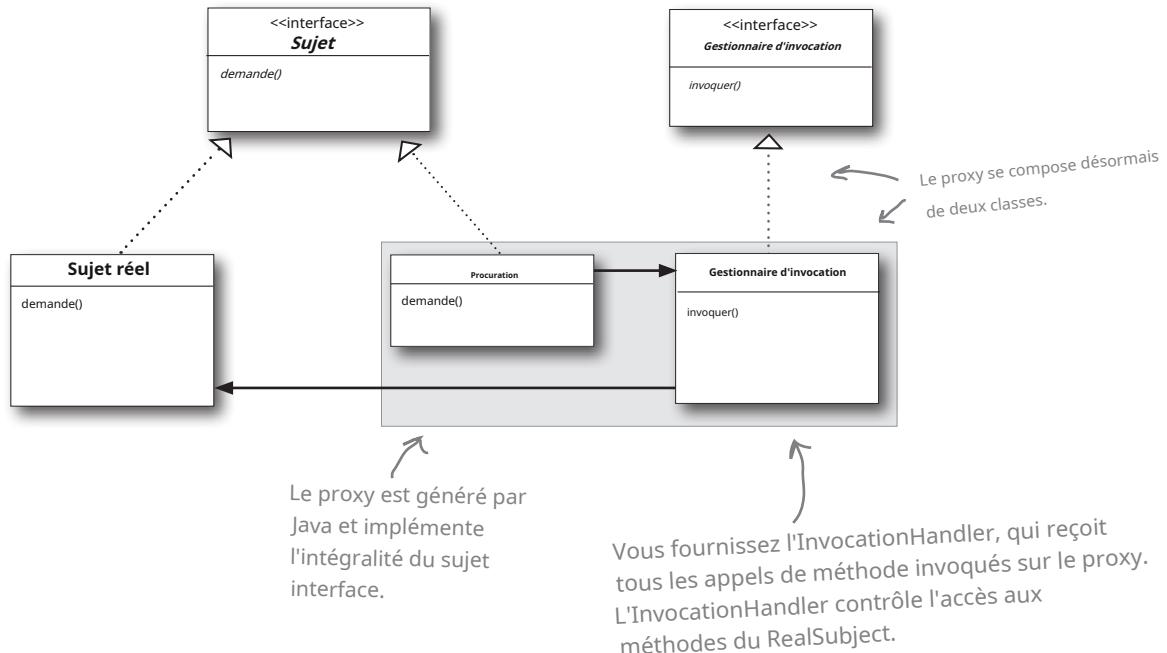
C'est comme un proxy, agissant comme un vrai, alors qu'en fait, vous ne faites que remplacer les objets qui font le vrai travail. Vous savez, je suis vraiment désolé pour vous.

Utilisation du proxy de l'API Java pour créer un proxy de protection



Java possède son propre support proxy directement dans le package `java.lang.reflect`. Avec ce package, Java vous permet de créer une classe proxy à la volonté qui implémente une ou plusieurs interfaces et transmet les invocations de méthodes à une classe que vous spécifiez. Étant donné que la classe proxy réelle est créée au moment de l'exécution, nous appelons cette technologie Java une *proxy dynamique*.

Nous allons utiliser le proxy dynamique de Java pour créer notre prochaine implémentation de proxy (un proxy de protection), mais avant cela, examinons rapidement un diagramme de classes qui montre comment les proxys dynamiques sont assemblés. Comme la plupart des choses dans le monde réel, cela diffère légèrement de la définition classique du modèle :



Parce que Java crée la classe `Proxy` pour *toi*, vous avez besoin d'un moyen d'indiquer à la classe `Proxy` ce qu'elle doit faire. Vous ne pouvez pas mettre ce code dans la classe `Proxy` comme nous l'avons fait auparavant, car vous n'en implémentez pas directement une. Donc, si vous ne pouvez pas mettre ce code dans la classe `Proxy`, où le mettre ? Dans un `InvocationHandler`. Le travail de l'`InvocationHandler` est de répondre à tous les appels de méthode sur le proxy. Considérez l'`InvocationHandler` comme l'objet que le `Proxy` demande d'effectuer tout le vrai travail après avoir reçu les appels de méthode.

Ok, voyons comment utiliser le proxy dynamique...

protectionprocuration

Rencontres geek à Objectville

Chaque ville a besoin d'un service de rencontres, n'est-ce pas ? Vous avez relevé le défi et mis en place un service de rencontres pour Objectville. Vous avez également essayé d'innover en incluant une fonction « évaluation geek » où les participants peuvent évaluer le côté geek des autres (une bonne chose) — vous pensez que cela permet à vos clients de rester engagés et de rechercher des correspondances possibles ; cela rend également les choses beaucoup plus amusantes.

Votre service s'articule autour d'une interface Personne qui vous permet de paramétriser et d'obtenir des informations sur une personne :



```

Voici l'interface ; nous
passerons à l'implémentation
dans une seconde...
↓

interface publique Personne {

    Chaîne getName();
    Chaîne getGender();
    Chaîne getInterests();
    int getGeekRating();

    void setName(Chaîne nom); void
    setGender(Chaîne sexe); void
    setInterests(Chaîne intérêts); void
    setGeekRating(int note);

}

↑
Nous pouvons également définir
les mêmes informations via les
appels de méthode respectifs.

Ici, nous pouvons obtenir des
informations sur le nom de la
personne, son sexe, ses intérêts et
sa note Geek (1 à 10).
←

setGeekRating() prend un entier et
l'ajoute à la moyenne courante de
cette personne.
←

```

Voyons maintenant la mise en œuvre...

L'implémentation de la personne

```
classe publique PersonImpl implémente Person {
```

Nom de la chaîne ;

Sexe de la chaîne ;

Intérêts pour les cordes ;

note int;

int ratingCount = 0;

← Les variables d'instance.

```
chaîne publique getName() {
```

renvoyer le nom;

}

```
chaîne publique getGender() {
```

retourner le genre;

}

```
chaîne publique getInterests() {
```

intérêts de retour;

}

Toutes les méthodes getter ; elles renvoient chacune la variable d'instance appropriée...

```
public int getGeekRating() {
```

si (ratingCount == 0) renvoie 0 ; renvoie
(rating/ratingCount) ;

}

... sauf pour getGeekRating(),
qui calcule la moyenne des notes en divisant les notes par ratingCount.

```
public void setName(Chaîne nom) {
```

ceci.nom = nom;

}

```
public void setGender(String genre) {
```

this.gender = genre;

}

```
public void setInterests(String intérêts) {
```

this.intérêt = intérêts;

}

Et voici toutes les méthodes setter,
qui définissent la variable d'instance correspondante.

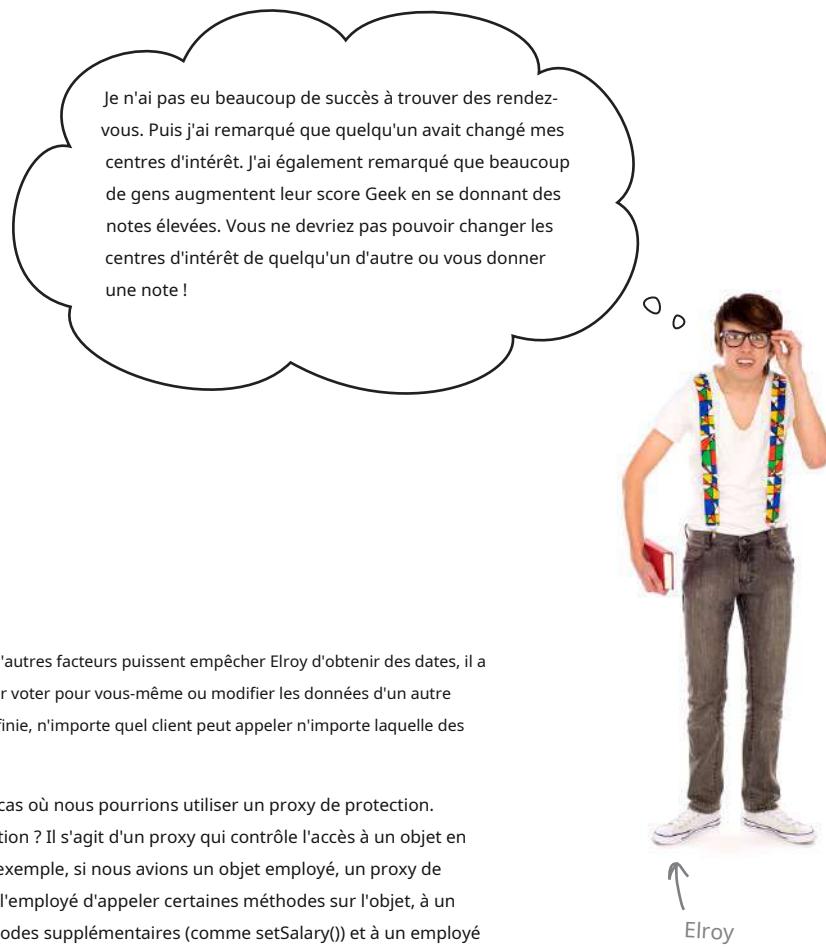
```
public void setGeekRating(int note) {
```

ceci.note += note ;
noteCount++;

}

Enfin, la méthode setGeekRating()
incrémentera le ratingCount total et ajoute la note au total cumulé.

}



Bien que nous soupçonnions que d'autres facteurs puissent empêcher Elroy d'obtenir des dates, il a raison : vous ne devriez pas pouvoir voter pour vous-même ou modifier les données d'un autre client. La façon dont Person est définie, n'importe quel client peut appeler n'importe laquelle des méthodes.

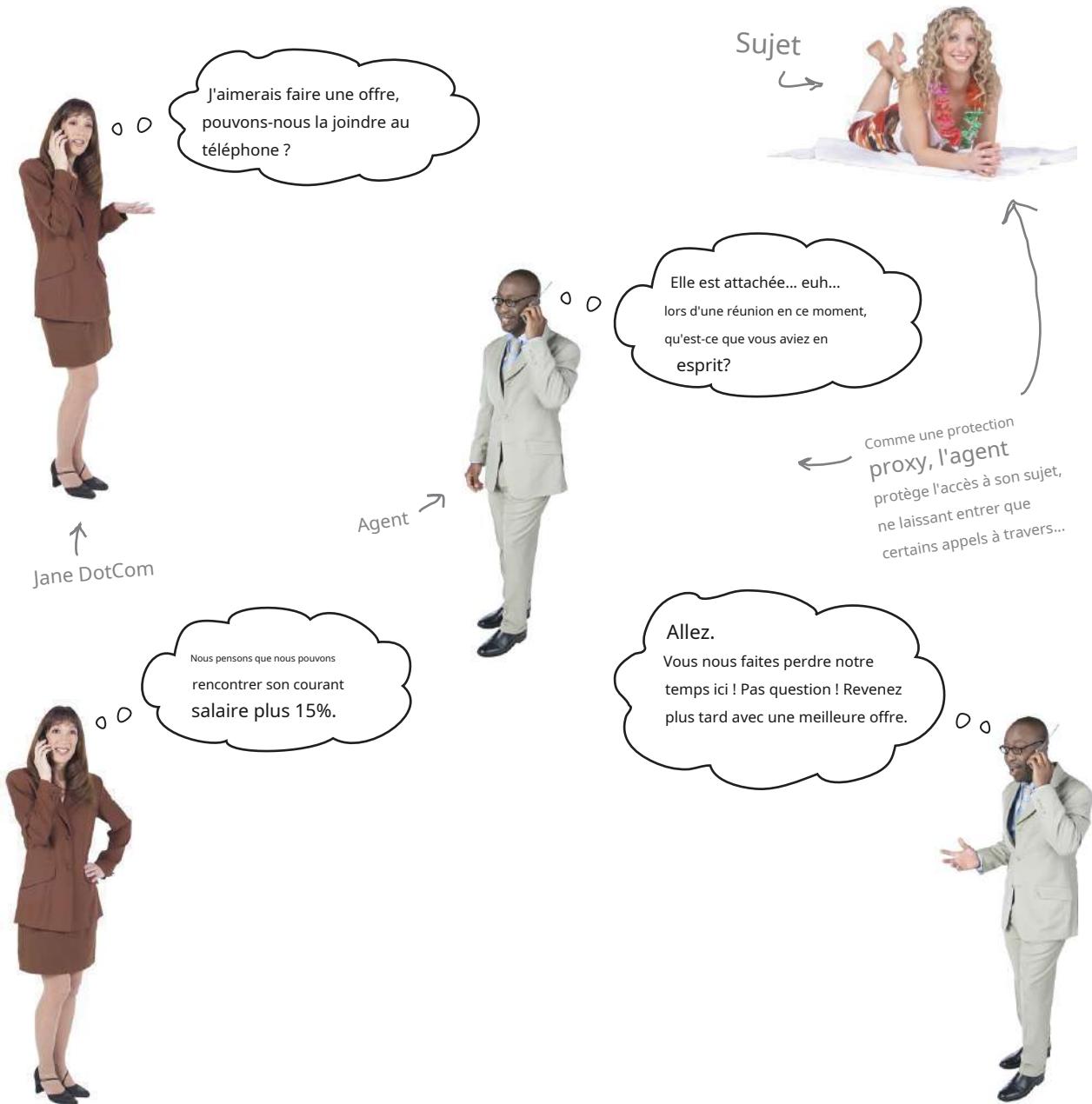
Il s'agit d'un parfait exemple de cas où nous pourrions utiliser un proxy de protection. Qu'est-ce qu'un proxy de protection ? Il s'agit d'un proxy qui contrôle l'accès à un objet en fonction des droits d'accès. Par exemple, si nous avions un objet employé, un proxy de protection pourrait permettre à l'employé d'appeler certaines méthodes sur l'objet, à un responsable d'appeler des méthodes supplémentaires (comme `setSalary()`) et à un employé des ressources humaines d'appeler n'importe quelle méthode sur l'objet.

Dans notre service de rencontres, nous voulons nous assurer qu'un client peut définir ses propres informations tout en empêchant les autres de les modifier. Nous voulons également permettre exactement le contraire avec les notes Geek : nous voulons que les autres clients puissent définir la note, mais pas ce client en particulier. Nous avons également un certain nombre de méthodes getter dans Person, et comme aucune d'entre elles ne renvoie d'informations privées, n'importe quel client devrait pouvoir les appeler.



Drame de cinq minutes : protéger les sujets

La bulle Internet semble être un lointain souvenir. C'était l'époque où il suffisait de traverser la rue pour trouver un emploi meilleur et mieux payé. Même les agents de développeurs de logiciels étaient à la mode...



Vue d'ensemble : créer un proxy dynamique pour la personne

Nous avons quelques problèmes à résoudre : les clients ne devraient pas pouvoir modifier leur propre évaluation Geek et les clients ne devraient pas pouvoir modifier les informations personnelles des autres clients. Pour résoudre ces problèmes, nous allons créer deux proxys : un pour accéder à votre propre objet Person et un pour accéder à l'objet Person d'un autre client. De cette façon, les proxys peuvent contrôler les demandes qui peuvent être faites dans chaque cas.

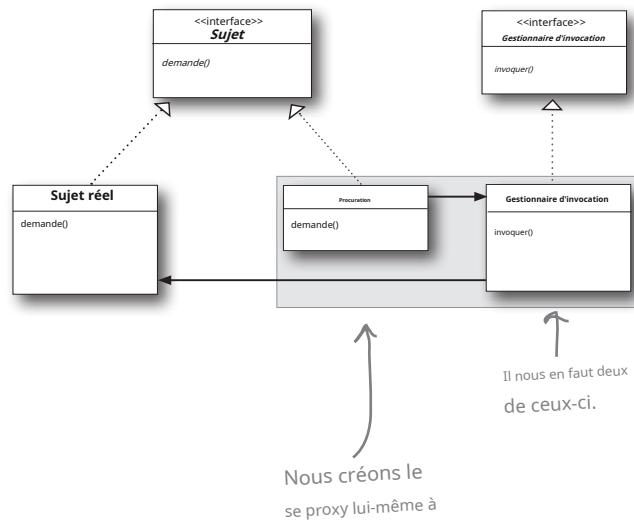
Pour créer ces proxys, nous allons utiliser le proxy dynamique de l'API Java que vous avez vu quelques pages plus haut. Java va créer deux proxys pour nous ; tout ce que nous devons faire est de fournir les gestionnaires qui savent quoi faire lorsqu'une méthode est invoquée sur le proxy.

Souvenez-vous de ce diagramme quelques pages en arrière...

Première étape :

Créer deux **Gestionnaires d'invocation**.

Les InvocationHandlers implémentent le comportement du proxy. Comme vous le verrez, Java se chargera de créer la classe et l'objet proxy réels ; nous devons simplement fournir un gestionnaire qui sait quoi faire lorsqu'une méthode est appelée dessus.



Deuxième étape :

Écrivez le code qui crée les proxys dynamiques.

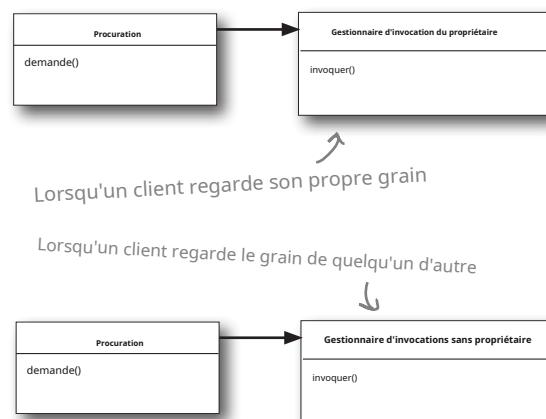
Nous devons écrire un peu de code pour générer la classe proxy et l'instancier. Nous allons parcourir ce code dans quelques instants.

Troisième étape :

Enveloppez n'importe quel objet Person avec le proxy approprié.

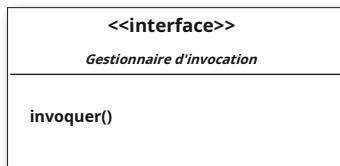
Lorsque nous devons utiliser un objet Person, soit il s'agit de l'objet du client lui-même (dans ce cas, nous l'appellerons le « propriétaire »), soit il s'agit d'un autre utilisateur du service que le client consulte (dans ce cas, nous l'appellerons « non-propriétaire »).

Dans les deux cas, nous créons le proxy approprié pour la personne.



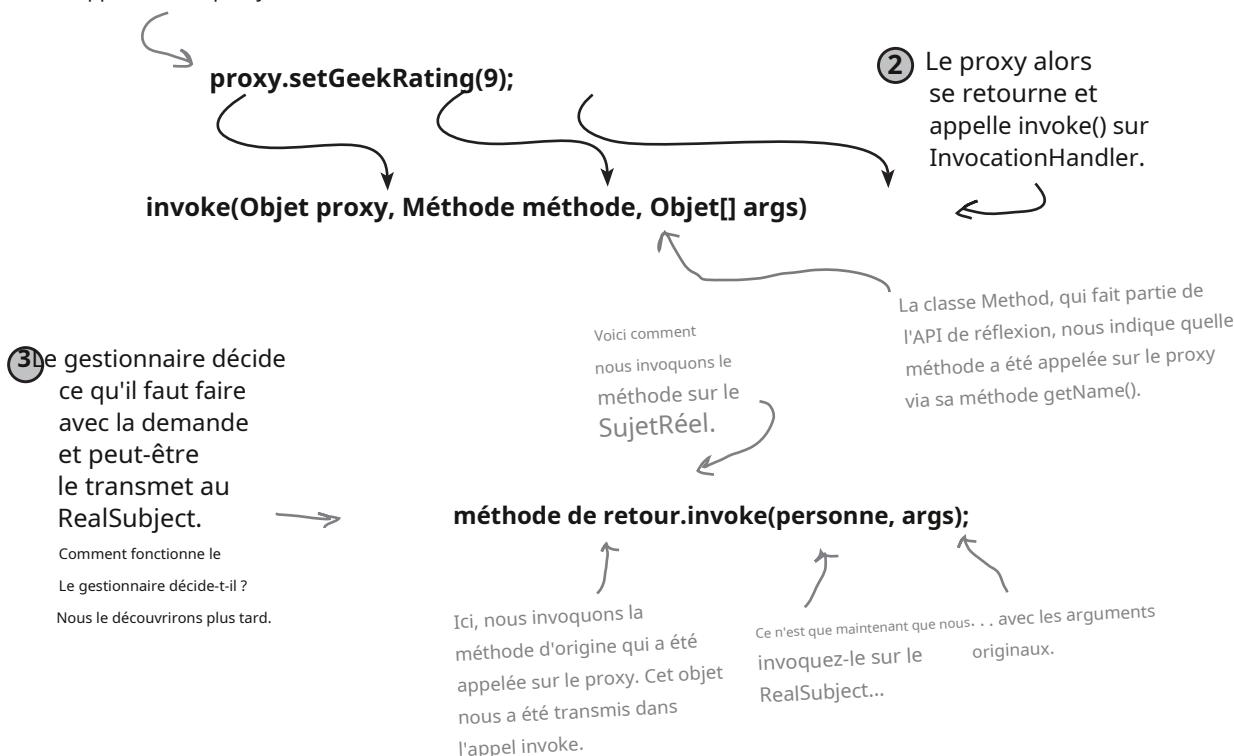
Première étape : création de gestionnaires d'invocation

Nous savons que nous devons écrire deux gestionnaires d'invocation, un pour le propriétaire et un pour le non-propriétaire. Mais que sont les gestionnaires d'invocation ? Voici comment les considérer : lorsqu'un appel de méthode est effectué sur le proxy, le proxy transmet cet appel à votre gestionnaire d'invocation, mais pas en appelant la méthode correspondante du gestionnaire d'invocation. Alors, qu'appelle-t-il ? Jetez un œil à l'interface InvocationHandler :



Il n'y a qu'une seule méthode, `invoke()`, et quelles que soient les méthodes appelées sur le proxy, c'est la méthode `invoke()` qui est appelée sur le gestionnaire. Voyons comment cela fonctionne :

- ① Disons que la méthode `setGeekRating()` est appelée sur le proxy.



Création de gestionnaires d'invocation, suite...

Lorsque invoke() est appelé par le proxy, comment savez-vous quoi faire avec l'appel ? En règle générale, vous examinerez la méthode qui a été appelée sur le proxy et prendrez des décisions en fonction du nom de la méthode et éventuellement de ses arguments. Implémentons OwnerInvocationHandler pour voir comment cela fonctionne :

InvocationHandler fait partie du package
java.lang.reflect, nous devons donc l'importer.

```
importer java.lang.reflect.*;
```

Tous les gestionnaires
d'invocation implémentent le
Interface InvocationHandler.

classe publique OwnerInvocationHandler implémente InvocationHandler {

```
Personne personne;
```

```
public OwnerInvocationHandler(Personne personne) {  
    cette.personne = personne;  
}
```

```
Objet public invoke(Object proxy, Method method, Object[] args)  
génère IllegalAccessException {
```

Nous avons dépassé le
RealSubject dans le
constructeur et nous
garder une référence à cela.

Voici l'invocation()
méthode qui obtient
appelé à chaque fois qu'une
méthode est invoquée
sur le proxy.

```
essayer {  
    si (méthode.getName().startsWith("get")) {  
        méthode de retour.invoke(personne, args);  
    } sinon si (méthode.getName().equals("setGeekRating")) {  
        lancer une nouvelle IllegalAccessException();  
    } sinon si (méthode.getName().startsWith("set")) {  
        méthode de retour.invoke(personne, args);  
    }  
} catch (InvocationTargetException e) {  
    e.printStackTrace();  
}  
retourner null;
```

Si la méthode est un getter,
nous allons de l'avant et
l'invoquons sur le sujet réel.

Sinon, s'il s'agit de
setGeekRating()
méthode que nous interdisons
en le jetant
Exception d'accès illégal.

Cela se produira
si le vrai sujet
génère une exception.
Parce que nous sommes le
propriétaire, toute autre
méthode définie convient et
nous allons de l'avant et
l'invoquons sur le sujet réel.

Si une autre méthode est appelée,
nous allons simplement renvoyer null
plutôt que de prendre un risque.



Le NonOwnerInvocationHandler fonctionne exactement comme le OwnerInvocationHandler, sauf qu'il *permet* d'appeler setGeekRating() et il *interdit* tous les autres appels à toute autre méthode définie. Allez-y et écrivez ce gestionnaire vous-même :

Deuxième étape : création de la classe Proxy et instantiation de l'objet Proxy

Il ne nous reste plus qu'à créer dynamiquement la classe Proxy et à instancier l'objet proxy.

Commençons par écrire une méthode qui prend un objet Person et sait comment créer un proxy propriétaire pour celui-ci. Autrement dit, nous allons créer le type de proxy qui transmet ses appels de méthode à OwnerInvocationHandler. Voici le code :

```

    Cette méthode prend un objet Person (le sujet réel) et
    renvoie un proxy pour celui-ci. Étant donné que le proxy a
    la même interface que le sujet, nous renvoyons un objet
    Person.

    Ce code crée le proxy.
    C'est un code vraiment
    moche, alors étudions-
    le attentivement.

    Pour créer un proxy, nous utilisons la
    méthode statique newProxyInstance()
    sur la classe Proxy.

    Nous lui passons le chargeur de classe pour notre sujet...
    ... et l'ensemble des interfaces que
    le proxy doit implémenter...
    ... et un gestionnaire d'invocation, dans ce
    cas notre OwnerInvocationHandler.

    Nous passons le sujet réel au constructeur du gestionnaire
    d'invocation. Si vous revenez deux pages en arrière, vous
    verrez que c'est ainsi que le gestionnaire accède au sujet
    réel.
}

Personne getOwnerProxy(Personne personne) {
    retourner (Personne) Proxy.newProxyInstance(
        personne.getClass().getClassLoader(),
        personne.getClass().getInterfaces(), nouveau
        OwnerInvocationHandler(personne));
}

```



Bien que cela soit un peu compliqué, la création d'un proxy dynamique n'est pas très compliquée. Pourquoi ne pas écrire `getNonOwnerProxy()`, qui renvoie un proxy pour `NonOwnerInvocationHandler` :

Allez plus loin : pouvez-vous écrire une méthode appelée `getProxy()` qui prend un gestionnaire et une personne et renvoie un proxy qui utilise ce gestionnaire ?

Tester le service de matchmaking

Testons le service de matchmaking et voyons comment il contrôle l'accès aux méthodes de définition en fonction du proxy utilisé.

```

classe publique MatchMakingTestDrive {
    // variables d'instance ici

    public static void main(String[] args) {
        test.drive();

    }

    public MatchMakingTestDrive() {
        initialiserDatabase();
    }

    lecteur public void() {
        Personne joe = getPersonFromDatabase("Joe Javabean");
        Personne ownerProxy = getOwnerProxy(joe);
        System.out.println("Le nom est " + ownerProxy.getName());
        ownerProxy.setInterests("bowling, Go");
        System.out.println("Intérêts définis à partir du proxy du propriétaire");
        essayez {
            propriétaireProxy.setGeekRating(10);
        catch (Exception e) {
            System.out.println("Impossible de définir la note à partir du proxy propriétaire");
        }
        System.out.println("La note est " + ownerProxy.getGeekRating());

        Personne nonOwnerProxy = getNonOwnerProxy(joe);
        System.out.println("Le nom est " + nonOwnerProxy.getName());
        essayez {
            nonOwnerProxy.setInterests("bowling, Go");
        catch (Exception e) {
            System.out.println("Impossible de définir les intérêts à partir d'un proxy non propriétaire");
        }
        nonOwnerProxy.setGeekRating(3);
        System.out.println("Note définie à partir d'un proxy non propriétaire");
        System.out.println("La note est " + nonOwnerProxy.getGeekRating());
    }

    // d'autres méthodes comme getOwnerProxy et getNonOwnerProxy ici
}

```

La méthode main() crée simplement le test drive et appelle sa méthode drive() pour faire avancer les choses.

Le constructeur initialise notre base de données de personnes dans le service de matchmaking.

Récupérons une personne dans la base de données...

... et créer un proxy propriétaire.

Appeler un getter...

... et puis un setter.

Et puis essayez de changer la note.

Cela ne devrait pas fonctionner !

Créez maintenant un proxy non propriétaire...

... et appeler un getter...

... suivi d'un setter.

Cela ne devrait pas fonctionner !

Ensuite, essayez de définir la notation.

Cela devrait fonctionner !

Exécution du code...

```
Aide de la fenêtre d'édition de fichier Born2BDynamic
% java MatchMakingTestDrive
Le nom est Joe Javabean
Intérêts définis à partir du proxy du propriétaire Impossible
de définir une note à partir du proxy du propriétaire La note
est de 7
Notre proxy propriétaire permet
d'obtenir et de paramétrer,
sauf pour la note
Geek.

Mon nom est Joe Javabean
Impossible de définir les intérêts à partir d'un proxy non propriétaire
Évaluation définie à partir d'un proxy non propriétaire
La note est de 5
%
Notre proxy NonOwner permet
d'obtenir uniquement, mais aussi
permet aux appels de définir la
note Geek.

La nouvelle note est la moyenne de la note précédente, 7, et
de la valeur définie par le proxy NonOwner, 3.
```

there are no
Dumb Questions

Q:

Alors, quel est exactement l'aspect « dynamique » des proxys dynamiques ? Est-ce que j'instancie le proxy et le configure sur un gestionnaire au moment de l'exécution ?

UN:

Non, le proxy est dynamique car sa classe est créée au moment de l'exécution. Pensez-y : avant que votre code ne s'exécute, il n'y a pas de classe proxy ; il est créé à la demande à partir de l'ensemble des interfaces que vous lui transmettez.

Q:

Mon InvocationHandler semble être un proxy très étrange ; il n'implémente aucune des méthodes de la classe qu'il proxy.

UN:

C'est parce que InvocationHandler n'est pas un proxy, mais une classe vers laquelle le proxy se dirige pour gérer les appels de méthode. Le proxy lui-même est créé dynamiquement au moment de l'exécution par la méthode statique Proxy.newProxyInstance().

Q:

Existe-t-il un moyen de savoir si une classe est une classe proxy ?

UN:

Oui. La classe Proxy possède une méthode statique appelée isProxyClass(). L'appel de cette méthode avec une classe renverra true si la classe est une classe proxy dynamique. À part cela, la classe proxy agira comme toute autre classe qui implémente un ensemble particulier d'interfaces.

Q:

Existe-t-il des restrictions sur les types d'interfaces que je peux transmettre à newProxyInstance() ?

UN:

Oui, il y en a quelques-unes. Tout d'abord, il convient de souligner que nous transmettons toujours à newProxyInstance() un tableau d'interfaces : seules les interfaces sont autorisées, pas les classes. Les principales restrictions sont que toutes les interfaces non publiques doivent provenir du même package. Vous ne pouvez pas non plus avoir d'interfaces avec des noms de méthodes contradictoires (c'est-à-dire deux interfaces avec une méthode avec la même signature). Il existe également quelques autres nuances mineures, donc à un moment donné, vous devriez jeter un œil aux petits caractères sur les proxys dynamiques dans la javadoc.



Associez chaque motif à sa description :

Modèle

Description

Décorateur

Enveloppe un autre objet et lui fournit une interface différente.

Façade

Enveloppe un autre objet et fournit un comportement supplémentaire pour cela.

Procuration

Enveloppe un autre objet pour en contrôler l'accès.

Adaptateur

Enveloppe un ensemble d'objets pour simplifier leur interface.

Le zoo par procuration

Bienvenue au zoo d'Objectville !

Vous connaissez maintenant les proxys distants, virtuels et de protection, mais dans la nature, vous allez voir de nombreuses mutations de ce modèle. Ici, dans le coin Proxy du zoo, nous avons une belle collection de modèles de proxy sauvages que nous avons capturés pour votre étude.

Notre travail n'est pas terminé ; nous sommes sûrs que vous verrez d'autres variantes de ce modèle dans le monde réel, alors aidez-nous à cataloguer davantage de proxys. Jetons un œil à la collection existante :



Pare-feu proxy
contrôle l'accès à un
ensemble de réseau
ressources, protection
le sujet des « mauvais » clients.

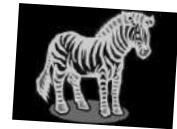


Habitat : souvent observé à l'emplacement
des systèmes de pare-feu d'entreprise.

Aidez-nous à trouver un habitat



Proxy de référence intelligent
fournit des actions supplémentaires
chaque fois qu'un sujet
est référencé, comme compter
le nombre de références à
un objet.



Proxy de mise en cache
fournit stockage temporaire pour
résultats d'exploitation
qui sont chers. Il



Habitat : souvent observé dans les proxys de serveurs Web ainsi
que dans les systèmes de gestion de contenu et de publication.

peut également permettre à plusieurs clients de
partager les résultats pour réduire la latence de
calcul ou du réseau.

Proxy de synchronisation
fournit un accès sécurisé à un sujet à partir de plusieurs threads.



On le voit traîner dans les collections, où il contrôle l'accès synchronisé à un ensemble sous-jacent d'objets dans un environnement multithread.

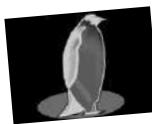
Aidez-nous à trouver un habitat



Proxy masquant la complexité
masque la complexité et
contrôle l'accès à un
ensemble complexe de
classes. Ceci est parfois appelé
le proxy de façade pour des raisons évidentes. Le
proxy de masquage de complexité diffère de



le modèle de façade dans lequel le proxy
contrôle l'accès, tandis que le modèle de façade
fournit simplement une interface alternative.



Proxy de copie sur écriture
contrôle la copie d'un
objet en différant la
copie d'un

objet jusqu'à ce qu'il soit requis
par un client. Il s'agit d'une
variante du proxy virtuel.



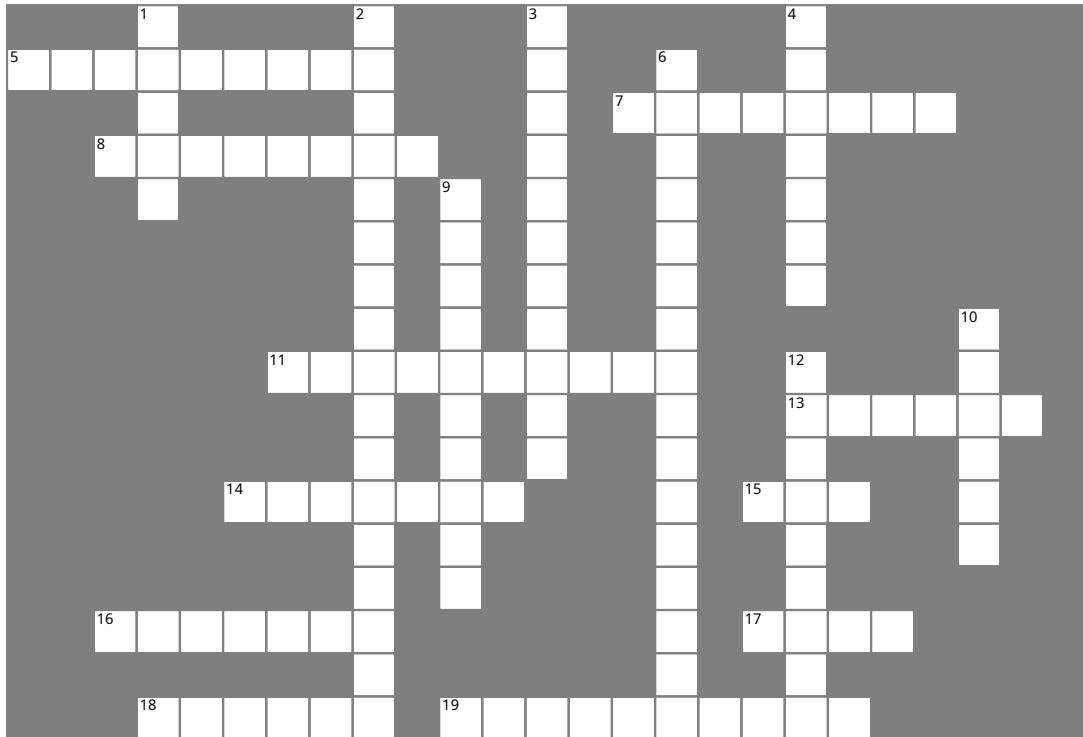
Habitat : observé à proximité de
CopyOnWriteArrayList de Java.

Notes de terrain : veuillez ajouter ici vos observations d'autres proxys dans la nature :



Mots croisés sur les modèles de conception

Ce fut un LONG chapitre. Pourquoi ne pas vous détendre en faisant des mots croisés avant qu'il ne se termine ?



À TRAVERS

5. Groupe de couvertures du premier album affiché (deux mots).
7. Proxy couramment utilisé pour les services Web (deux mots).
8. Dans RMI, l'objet qui prend les requêtes réseau côté service.
11. Proxy qui protège les appels de méthode contre les appelants non autorisés.
13. Groupe qui a réalisé l'album MCMXC aD
14. Une classe proxy _____ est créée lors de l'exécution.
15. Lieu pour en savoir plus sur les nombreuses variantes de proxy.
16. Le visualiseur d'album a utilisé ce type de proxy.
17. Dans RMI, le proxy s'appelle ainsi.
18. Nous en avons pris un pour apprendre le RMI.
19. Pourquoi Elroy n'a pas pu obtenir de rendez-vous.

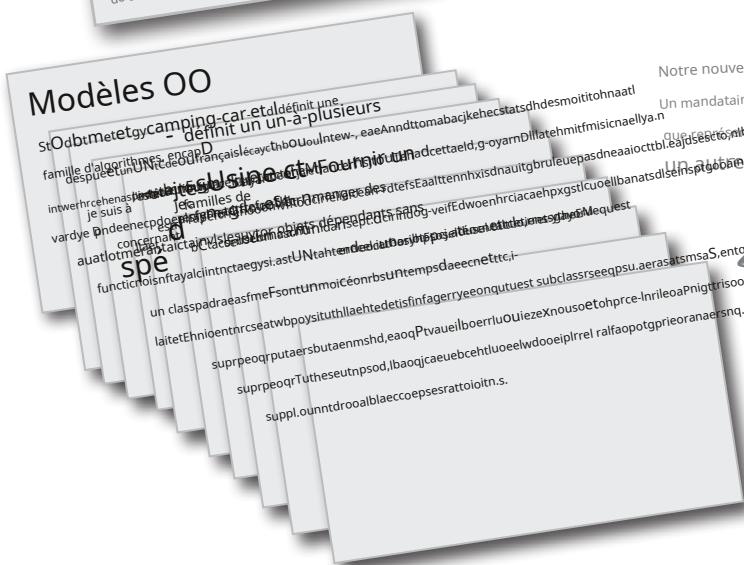
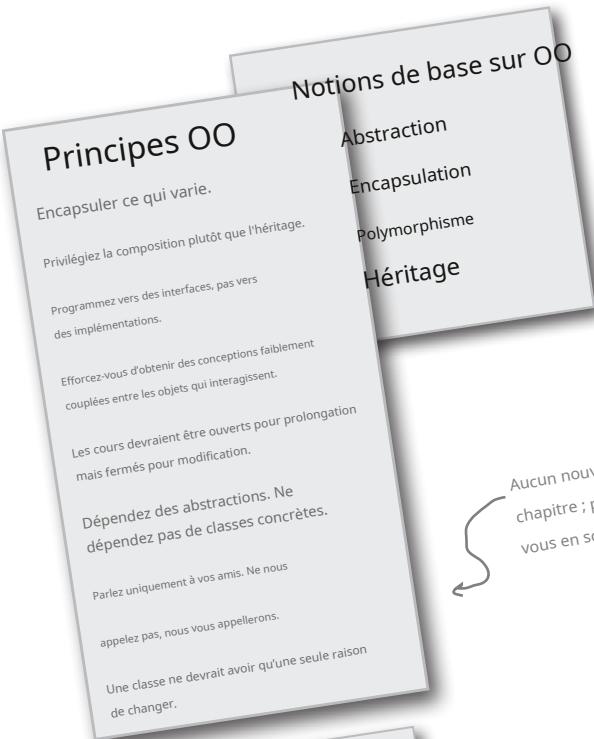
VERS LE BAS

1. Objectville Matchmaking est pour _____.
2. Le proxy dynamique de Java transmet toutes les requêtes à ceci (deux mots).
3. Cet utilitaire agit comme un service de recherche pour RMI.
4. Proxy qui remplace les objets coûteux.
6. La télécommande _____ a été utilisée pour implémenter le moniteur du distributeur de boules de gomme (deux mots).
9. L'agent développeur de logiciels était ce genre de proxy.
10. Notre première erreur : le rapport sur le distributeur de boules de gomme n'était pas _____.
12. Similaire au proxy, mais avec un objectif différent.



Des outils pour votre boîte à outils de conception

Votre boîte à outils de conception est presque pleine ; vous êtes préparé à presque tous les problèmes de conception qui se présentent à vous.



BULLET POINTS

- f Le modèle de proxy fournit un représentant pour un autre objet afin de contrôler l'accès du client à celui-ci. Il existe plusieurs façons de gérer cet accès.
 - f Un proxy distant gère l'interaction entre un client et un objet distant.
 - f Un proxy virtuel contrôle l'accès à un objet dont l'instanciation est coûteuse.
 - f Un proxy de protection contrôle l'accès aux méthodes d'un objet en fonction de l'appelant.
 - f Il existe de nombreuses autres variantes du modèle Proxy, notamment les proxys de mise en cache, les proxys de synchronisation, les proxys de pare-feu, les proxys de copie sur écriture, etc.
 - f Proxy est structurellement similaire à Decorator, mais les deux modèles diffèrent dans leur objectif.
 - f Le modèle Decorator ajoute un comportement à un objet, tandis que le proxy contrôle l'accès.
 - f La prise en charge intégrée de Java pour Proxy peut créer une classe proxy dynamique à la demande et répartir tous les appels vers un gestionnaire de votre choix.
 - f Comme tout wrapper, les proxys augmenteront le nombre de classes et d'objets dans vos conceptions.



Exercise Solution

Le NonOwnerInvocationHandler fonctionne exactement comme le OwnerInvocationHandler, sauf qu'il permet appelle setGeekRating() et il interdit appels à toute autre méthode définie. Voici notre solution :

```
importer java.lang.reflect.*;  
  
classe publique NonOwnerInvocationHandler implémente InvocationHandler {  
    Personne personne;  
  
    public NonOwnerInvocationHandler(Personne personne) {  
        cette.personne = personne;  
    }  
  
    Objet public invoke(Object proxy, Method method, Object[] args)  
        génère IllegalAccessException {  
  
        essayer {  
            si (méthode.getName().startsWith("get")) {  
                méthode de retour.invoke(personne, args);  
            } sinon si (méthode.getName().equals("setGeekRating")) {  
                méthode de retour.invoke(personne, args);  
            } sinon si (méthode.getName().startsWith("set")) {  
                lancer une nouvelle IllegalAccessException();  
            }  
        } catch (InvocationTargetException e) {  
            e.printStackTrace();  
        }  
        retourner null;  
    }  
}
```



Solution du puzzle de conception

La classe ImageProxy semble avoir deux états contrôlés par des instructions conditionnelles. Pouvez-vous penser à un autre modèle qui pourrait nettoyer ce code ? Comment reconcevriez-vous ImageProxy ?

Utilisez le modèle d'état : implémentez deux états, ImageLoaded et ImageNotLoaded. Placez ensuite le code des instructions if dans leurs états respectifs. Commencez dans l'état ImageNotLoaded, puis passez à l'état ImageLoaded une fois l'ImageIcon récupérée.



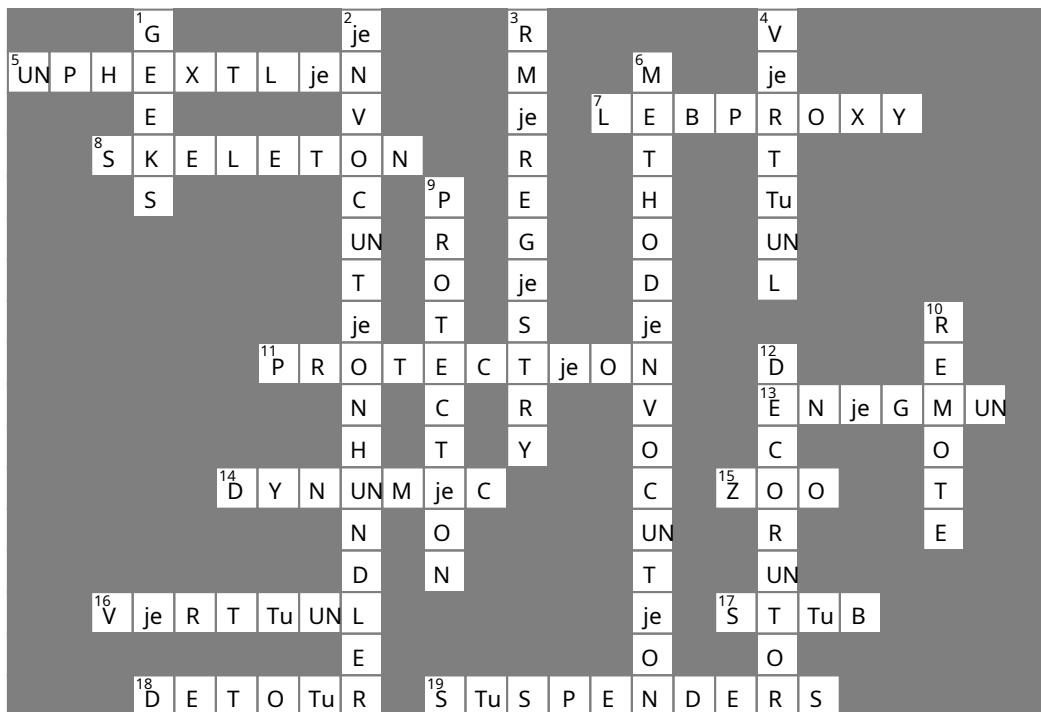
Bien que cela soit un peu compliqué, la création d'un proxy dynamique n'est pas très compliquée. Pourquoi n'écrivez-vous pas `getNonOwnerProxy()`, qui renvoie un proxy pour le `NonOwnerInvocationHandler`? Voici notre solution :

```
Personne getNonOwnerProxy(Personne personne) {

    retourner (Personne) Proxy.newProxyInstance(
        personne.getClass().getClassLoader(),
        personne.getClass().getInterfaces(), nouveau
        NonOwnerInvocationHandler(personne));
}
```



Solution de mots croisés sur les modèles de conception





Associez chaque motif à sa description :

Modèle	Description
Décorateur	Enveloppe un autre objet et lui fournit une interface différente.
Façade	Enveloppe un autre objet et fournit un comportement supplémentaire pour cela.
Procuration	Enveloppe un autre objet pour en contrôler l'accès.
Adaptateur	Enveloppe un ensemble d'objets pour simplifier leur interface.



Prêt à cuire Code

Le code pour la visionneuse de couverture d'album

```
paquet headfirst.designpatterns.proxy.virtualproxy;

importer java.net.*;
importer java.awt.*;
importer java.awt.event.*;
importer javax.swing.*;
importer java.util.*;

classe publique ImageProxyTestDrive {
    Composant imageComposant;
    JFrame frame = new JFrame("Visionneuse de couverture d'album");
    JMenuBar menuBar;
    Menu JMenu;
    Hashtable<Chaîne, Chaîne> albums = nouveau Hashtable<Chaîne, Chaîne>();

    public static void main (String[] args) génère une exception {
        ImageProxyTestDrive testDrive = nouveau ImageProxyTestDrive();
    }

    public ImageProxyTestDrive() génère une exception {
        albums.put("Bar Bouddha","http://images.amazon.com/images/P/B00009XBYK.01.LZZZZZZZ.
jpg");
        albums.put("Ima","http://images.amazon.com/images/P/B000005IRM.01.LZZZZZZZ.jpg");
        albums.put("Karma","http://images.amazon.com/images/P/B000005DCB.01.LZZZZZZZ.
gif");
        albums.put("MCMXC aD","http://images.amazon.com/images/P/B000002URV.01.LZZZZZZZ.
jpg");
        albums.put("Northern Exposure","http://images.amazon.com/images/P/B000003SFN.01.
LZZZZZZZ.jpg");
        albums.put("Œuvres d'ambiance sélectionnées, vol. 2","http://images.amazon.com/images/P/
B000002MNZ.01.LZZZZZZZ.jpg");

        URL initialURL = new URL((String)albums.get("Œuvres ambiantes sélectionnées, vol. 2"));
        menuBar = new JMenuBar();
        menu = new JMenu("Albums favoris");
        menuBar.add(menu);
    }
}
```



Prêt à cuire
Code

Le code de la pochette de l'album
Téléspectateur, suite...

```
cadre.setJMenuBar(menuBar);

pour (Énumération e = albums.keys(); e.hasMoreElements();) {
    Nom de chaîne = (Chaîne)e.nextElement();
    JMenuItem menuItem = nouveau JMenuItem(nom);
    menu.add(menuItem);
    menuItem.addActionListener(événement -> {
        imageComponent.setIcon(
            nouveau ImageProxy(getAlbumUrl(event.getActionCommand())));
        cadre.repaint();
    });
}

// configurer le cadre et les menus

Icône icône = nouveau ImageProxy(initialURL); imageComponent =
nouveau ImageComponent(icône);
frame.getContentPane().add(imageComponent);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setSize(800,600);
cadre.setVisible(true);

}

URL getAlbumUrl (Nom de la chaîne) {
    essayer {
        renvoyer une nouvelle URL ((String)albums.get
        (name)); } catch (MalformedURLException e) {
        e.printStackTrace();
        retourner null;
    }
}
```



Prêt à cuire
Code

Le code de la pochette de l'album
Téléspectateur, suite...

```
paquet headfirst.designpatterns.proxy.virtualproxy;
```

```
importer java.net.*;
importer java.awt.*;
importer javax.swing.*;

la classe ImageProxy implémente Icon {
    volatile ImageIcon imageIcon; URL
    finale imageURL;
    Récupération de threadThread;
    booléen récupération = false;

    public ImageProxy(URL url) { imageURL = url; }

    public int getIconWidth() {
        si (imageIcon != null) {
            retourner imageIcon.getIconWidth(); }
        sinon {
            retourner 800;
        }
    }

    public int getIconHeight() {
        si (imageIcon != null) {
            retourner imageIcon.getIconHeight(); }
        sinon {
            retourner 600;
        }
    }

    synchronisé void setImageIcon(ImageIcon imageIcon) {
        ceci.imageIcon = imageIcon;
    }

    public void paintIcon(composant final c, graphiques g, int x, int y) {
        si (imageIcon != null) {
            imageIcon.paintIcon(c, g, x, y); } else {

            g.drawString("Chargement de la couverture de l'album, veuillez patienter...", x+300,
            y+190); if (!retrieving) {
                récupération = vrai;
```



Prêt à cuire
Code

Le code pour la visionneuse de couverture d'album,
suite...

```
retrievalThread = nouveau thread (nouveau runnable) {
    public void run() {
        essayer {
            setImageIcon(new ImageIcon(imageURL, "Couverture d'album"));
            c.repaint();
        } attraper (Exception e) {
            e.printStackTrace();
        }
    }
};

récupérationThread.start();
}

}

paquet headfirst.designpatterns.proxy.virtualproxy;

importer java.awt.*;
importer javax.swing.*;

classe ImageComponent étend JComponent {
    icône privée icône;

    public ImageComponent(Icon icône) {
        ceci.icon = icône;
    }

    public void setIcon(Icon icône) {
        ceci.icon = icône;
    }

    public void paintComponent(Graphiques g) {
        super.paintComponent(g); int w =
        icône.getIconWidth(); int h =
        icône.getIconHeight(); int x = (800 -
        w)/2;
        int y = (600 - h)/2;
        icône.paintIcon(ceci, g, x, y);
    }
}
```

12motifs composés

Motifs des motifs



Qui aurait pu imaginer que les modèles pourraient fonctionner ensemble ?

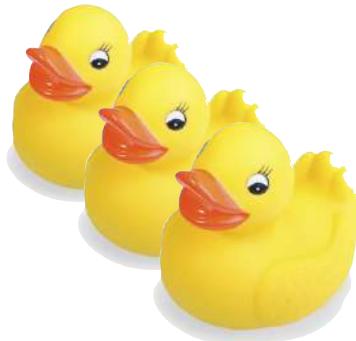
Vous avez déjà assisté aux causeries au coin du feu acrimonieuses (et vous n'avez même pas vu les pages Pattern Death Match que l'éditeur nous a forcé à supprimer du livre), alors qui aurait pensé que les motifs pouvaient réellement bien s'entendre ? Eh bien, croyez-le ou non, certaines des conceptions OO les plus puissantes utilisent plusieurs motifs ensemble. Préparez-vous à faire passer vos compétences en matière de motifs au niveau supérieur ; il est temps de passer aux motifs composés.

Travailler ensemble

L'une des meilleures façons d'utiliser les motifs est de les sortir de la maison afin qu'ils puissent interagir avec d'autres motifs. Plus vous utilisez des motifs, plus vous les verrez apparaître ensemble dans vos créations. Nous avons un nom spécial pour un ensemble de motifs qui fonctionnent ensemble dans une conception qui peut être appliquée à de nombreux problèmes : un *motif composé*. C'est vrai, nous parlons maintenant de motifs faits de motifs !

Vous trouverez de nombreux modèles composés utilisés dans le monde réel. Maintenant que vous avez ces modèles dans votre cerveau, vous verrez qu'ils ne sont en réalité que des modèles fonctionnant ensemble, ce qui les rend plus faciles à comprendre.

Nous allons commencer ce chapitre en revisitant nos sympathiques canards dans le simulateur de canards SimUDuck. Il est tout à fait normal que les canards soient là lorsque nous combinons des modèles ; après tout, ils nous ont accompagnés tout au long du livre et ils ont été de bons sportifs en participant à de nombreux modèles. Les canards vont vous aider à comprendre comment les modèles peuvent fonctionner ensemble dans la même solution. Mais ce n'est pas parce que nous avons combiné certains modèles que nous avons une solution qui peut être qualifiée de modèle composé. Pour cela, il doit s'agir d'une solution à usage général qui peut être appliquée à de nombreux problèmes. Ainsi, dans la deuxième moitié du chapitre, nous visiterons un *rôle/Modèle composé* : le modèle-vue-contrôleur, autrement connu sous le nom de MVC. Si vous n'avez jamais entendu parler de MVC, vous le saurez et vous découvrirez que MVC est l'un des modèles composés les plus puissants de votre boîte à outils de conception.



Les motifs sont souvent utilisés ensemble et combinés au sein d'une même solution de conception.

Un modèle composé combine deux ou plusieurs modèles dans une solution qui résout un problème récurrent ou général.

Retrouvailles entre canards

Comme vous l'avez déjà entendu, nous allons à nouveau travailler avec les canards. Cette fois, les canards vont vous montrer comment les modèles peuvent coexister et même coopérer au sein d'une même solution.

Nous allons reconstruire notre simulateur de canard à partir de zéro et lui donner des capacités intéressantes en utilisant un tas de modèles. Ok, commençons...

① Tout d'abord, nous allons créer une interface Quackable.

Comme nous l'avons dit, nous partons de zéro. Cette fois-ci, les canards vont implémenter une interface Quackable. De cette façon, nous saurons quels éléments du simulateur peuvent cancaner () - comme les canards colverts, les canards roux, les cris de canard, et nous pourrions même voir le canard en caoutchouc revenir furtivement.

```
interface publique Quackable {
    public void charlatan();
}
```

Les charlatans n'ont besoin que d'une seule chose bien faire : cancaner !

② Maintenant, certains canards implémentent Quackable

À quoi sert une interface sans quelques classes pour l'implémenter ? Il est temps de créer des canards en béton (mais pas du genre « art de pelouse », si vous voyez ce que nous voulons dire).

```
la classe publique MallardDuck implémente Quackable {
    public void charlatan() {
        System.out.println("Coin-coin");
    }
}
```

Votre norme
Canard colvert.

```
la classe publique RedheadDuck implémente Quackable {
    public void charlatan() {
        System.out.println("Coin-coin");
    }
}
```

Nous devons avoir une certaine variation d'espèces si nous voulons que ce soit un simulateur intéressant.

Ce ne serait pas très amusant si nous n'ajoutions pas également d'autres types de canards.

Vous vous souvenez de la dernière fois ? Nous avions des appeaux à canard (ces objets que les chasseurs utilisent, ils cancent vraiment) et des canards en caoutchouc.

la classe publique DuckCall implémente Quackable {

public void charlatan() {

Système.out.println("Kwak");

}

la classe publique RubberDuck implémente Quackable {

public void charlatan() {

System.out.println("Grincement");

}

}



Un appel de canard qui cancane mais qui ne ressemble pas vraiment à un vrai.



Un canard en caoutchouc qui fait un couinement quand il cancane.

③

Ok, nous avons nos canards ; maintenant tout ce dont nous avons besoin est un simulateur.

Créons un simulateur qui crée quelques canards et s'assure que leurs charlatans fonctionnent...

classe publique DuckSimulator {

public static void main(String[] args) {

DuckSimulator simulateur = nouveau DuckSimulator();

simulateur.simuler();

}



Voici notre méthode main()
pour tout faire fonctionner.



Nous créons un simulateur et
l'appelons ensuite
méthode simuler().

void simuler() {

Canard colvert cancanard ...



Nous avons besoin de quelques
canards, alors ici nous en créons
un de chaque Quackable...

System.out.println("\nSimulateur de canard");

simuler(canard colvert);

simuler(redheadDuck);

simuler(duckCall);

simuler(rubberDuck);

}



... puis nous simulons
chacun d'eux.



Ici, nous surchargeons la méthode
simuler() pour simuler un seul canard.

void simuler(canard canaille) {

canard.coin-coin();

}



Ici, nous laissons le polymorphisme faire sa magie : quel
que soit le type de Quackable transmis, la méthode
simule() lui demande de cancaner.

Pas trop excitant pour le moment, mais nous n'avons pas ajouté de modèles !



Fenêtre d'édition de fichier Aide ItBetterGetBetterThanThis

% Java DuckSimulator

Simulateur de canard

Charlatan
Charlatan
Kwak
Grincer

Ils implémentent tous la même interface Quackable, mais leurs implémentations leur permettent de cancaner à leur manière.

Il semble que tout fonctionne ; jusqu'à présent, tout va bien.

④ Quand les canards sont là, les oies ne peuvent pas être loin.

Là où il y a un oiseau aquatique, il y en a probablement deux. Voici une classe d'oies qui traîne dans le simulateur.

```
classe publique Oie {
    public void klaxonner() {
        System.out.println("Klaxonner");
    }
}
```

Une oie est un klaxon,
pas un charlatan.




Disons que nous voulons pouvoir utiliser une oie partout où nous voudrions utiliser un canard. Après tout, les oies font du bruit, les oies volent, les oies nagent. Pourquoi ne pouvons-nous pas avoir d'oies dans le simulateur ?

Quel modèle permettrait aux oies de se mélanger facilement aux canards ?

oieadaptateur

5

Nous avons besoin d'un adaptateur pour oie.

Notre simulateur s'attend à voir des interfaces Quackable. Comme les oies ne sont pas des cancaniers (elles sont des klaxonneuses), nous pouvons utiliser un adaptateur pour adapter une oie à un canard.

**la classe publique GooseAdapter implémente Quackable {
Oie oie;**

```
public GooseAdapter(oie oie) {  
    cette.oie = oie;  
}
```

N'oubliez pas, un adaptateur implémente l'interface cible, qui dans ce cas est Quackable.

```
public void charlatan() {  
    oie.honk();  
}  
}
```

Le constructeur prend l'oie que nous allons adapter.

Lorsque quack est appelé, l'appel est délégué à la méthode honk() de l'oie.

6

Désormais, les oies devraient également pouvoir jouer dans le simulateur.

Tout ce que nous devons faire est de créer un Goose et de l'envelopper dans un adaptateur qui implémente Quackable, et nous devrions être prêts à partir.

```
classe publique DuckSimulator {  
    public static void main(String[] args) {  
        DuckSimulator simulateur = nouveau DuckSimulator();  
        simulateur.simuler();  
    }  
}
```

```
void simuler() {  
    Canard colvert cancanard ...
```

Nous créons une oie qui agit comme un canard en enveloppant l'oie dans le GooseAdapter.

Oie cancanante = new GooseAdapter(new Goose());

System.out.println("\nDuck Simulator : avec adaptateur Goose ");

```
simuler(canard colvert);  
simuler(redheadDuck);  
simuler(duckCall);  
simuler(rubberDuck);  
simuler(oieCanard);  
}
```

Une fois l'oie emballée, nous pouvons la traiter comme les autres objets Quackable en forme de canard.

```
void simuler(canard canaille) {  
    canard.coin-coin();  
}  
}
```

⑦

Maintenant, essayons ceci rapidement...

Cette fois, lorsque nous exécutons le simulateur, la liste des objets passés à la méthode simuler() inclut une oie enveloppée dans un adaptateur canard. Le résultat ? Nous devrions voir des klaxons !

```
Fenêtre d'édition de fichier Aide GoldenEggs
% Java DuckSimulator
Simulateur de canard : avec adaptateur d'oie
Quack
Charlatan
Kwak
Grincer
Klaxonner
```

Voilà l'oie ! Maintenant l'oie peut cancaner avec les autres canards.

**Charlatanisme**

Les charlatans sont fascinés par tous les aspects du comportement cancanier. Une chose qu'ils ont toujours voulu étudier est le nombre total de cancanements émis par un troupeau de canards.

Comment pouvons-nous ajouter la possibilité de compter les cancanements des canards sans avoir à modifier les classes de canards ?

Pouvez-vous penser à un modèle qui pourrait aider ?



⑧ Nous allons rendre ces charlatans heureux et leur fournir quelques chiffres de charlatanisme.

Comment ? Créons un décorateur qui donne aux canards un nouveau comportement (le comportement de comptage) en les enveloppant d'un objet décorateur. Nous n'aurons pas du tout à modifier le code du canard.

```

QuackCounter est un décorateur.           Comme avec Adapter, nous devons
                                            implémenter l'interface cible.
                                            ↓
la classe publique QuackCounter implémente Quackable {
    Canard cancanant;
    nombre statique intOfQuacks;
    ↓
public QuackCounter (canard quackable) {
    ce.canard = canard;
    ↓
public void charlatan() {
    canard.coin-coin();
    nombreDeCharlatanismes++;
    ↓
public static int getQuacks() {
    retourner nombreDeQuacks;
    ↓
}

  Nous avons une variable d'instance
  pour conserver le charlatan que
  nous décorons.

  Et nous comptons TOUS les charlatans,
  nous utiliserons donc une variable
  statique pour garder une trace.

  Nous obtenons la référence au
  Quackable que nous décorons
  dans le constructeur.

  Lorsque quack() est appelé, nous déléguons l'appel
  au Quackable que nous décorons...
  ... puis on augmente le nombre de charlatans.

  Nous ajoutons une autre méthode au décorateur.
  Cette méthode statique renvoie simplement le
  nombre de charlatans qui se sont produits dans
  tous les Quackables.

```

⑨ Nous devons mettre à jour le simulateur pour créer des canards décorés.

Maintenant, nous devons envelopper chaque objet Quackable que nous instancions dans un décorateur QuackCounter. Si nous ne le faisons pas, nous aurons des canards qui courront partout en faisant des cancanements innombrables.

```

classe publique DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulateur = nouveau DuckSimulator();
        simulateur.simuler();
    }

    void simuler() {
        Canard colvert cancanard ...
        System.out.println("\nDuck Simulator : avec décorateur ");
        simulate(canard colvert);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
        simulate(oieCanard);
        System.out.println("Les canards cancaisaient " +
            QuackCounter.getQuacks() + " fois");
    }

    void simuler(canard canaille) {
        canard.coin-coin();
    }
}

```

Chaque fois que nous créons un Quackable, nous l'enveloppons d'un nouveau décorateur.

Le garde forestier nous a dit qu'il ne voulait pas compter les klaxons des oies, donc nous ne le décorons pas.

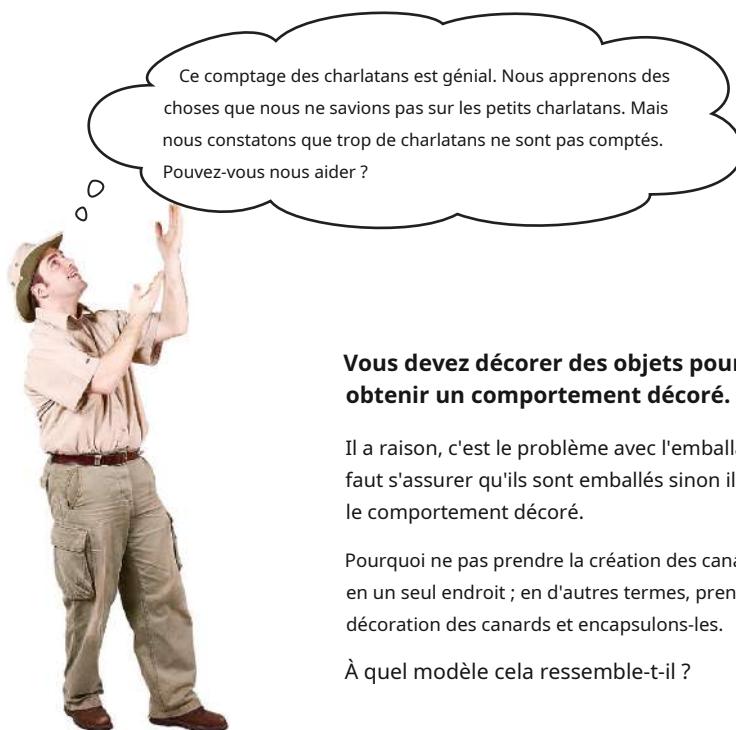
C'est ici que nous rassembler les charlatans comportement pour le Charlatans.

Rien ne change ici ; les objets décorés sont toujours des Quackables.

Voici le sortir!

Souviens-toi,
nous ne sommes pas
compter les oies.





You devez décorer des objets pour obtenir un comportement décoré.

Il a raison, c'est le problème avec l'emballage des objets : il faut s'assurer qu'ils sont emballés sinon ils n'obtiennent pas le comportement décoré.

Pourquoi ne pas prendre la création des canards et la localiser en un seul endroit ; en d'autres termes, prenons la création et la décoration des canards et encapsulons-les.

À quel modèle cela ressemble-t-il ?

⑩

Nous avons besoin d'une usine pour produire des canards !

D'accord, nous avons besoin d'un contrôle qualité pour nous assurer que nos canards sont emballés. Nous allons construire une usine entière juste pour les produire. L'usine doit produire une famille de produits composée de différents types de canards, nous allons donc utiliser le modèle d'usine abstraite.

Commençons par la définition de la classe `AbstractDuckFactory` :

```
classe abstraite publique AbstractDuckFactory {
```

```
    public abstrait Quackable createMallardDuck(); public
    abstrait Quackable createRedheadDuck(); public abstrait
    Quackable createDuckCall(); public abstrait Quackable
    createRubberDuck();
}
```

Nous définissons une usine abstraite que les sous-classes implémenteront pour créer différentes familles.

Chaque méthode crée un type de canard.

Ensuite, nous allons créer une usine qui crée des canards sans décorateurs, juste pour comprendre l'usine :

```
classe publique DuckFactory étend AbstractDuckFactory {

    public Quackable createMallardDuck() {
        renvoie un nouveau MallardDuck();
    }

    public Quackable createRedheadDuck() {
        renvoie le nouveau RedheadDuck();
    }

    public Quackable createDuckCall() {
        renvoie un nouveau DuckCall();
    }

    public Quackable createRubberDuck() {
        renvoie un nouveau RubberDuck();
    }
}
```

DuckFactory s'étend l'usine abstraite.

Chaque méthode crée un produit : un type particulier de Quackable. Le simulateur ne connaît pas le produit réel, il sait juste qu'il va recevoir un Quackable.

Créons maintenant l'usine que nous voulons vraiment, la CountingDuckFactory :

```
classe publique CountingDuckFactory étend AbstractDuckFactory {

    public Quackable createMallardDuck() {
        renvoie un nouveau QuackCounter(nouveau MallardDuck());
    }

    public Quackable createRedheadDuck() {
        renvoie un nouveau QuackCounter(nouveau RedheadDuck());
    }

    public Quackable createDuckCall() {
        renvoyer un nouveau QuackCounter(nouveau DuckCall());
    }

    public Quackable createRubberDuck() {
        renvoyer un nouveau QuackCounter(nouveau RubberDuck());
    }
}
```

Usine de comptage de canards étend également la usine abstraite.

Chaque méthode encapsule le Quackable avec le décorateur de comptage de quack. Le simulateur ne fera jamais la différence ; il récupère juste un Quackable. Mais maintenant, nos rangers peuvent être sûrs que tous les quacks sont comptés.

11 Configurons le simulateur pour utiliser l'usine.

Vous vous souvenez du fonctionnement d'Abstract Factory ? Nous créons une méthode polymorphe qui prend une factory et l'utilise pour créer des objets. En passant dans différentes factory, nous pouvons utiliser différentes familles de produits dans la méthode.

Nous allons modifier la méthode simuler() afin qu'elle prenne une usine et l'utilise pour créer des canards.

```

classe publique DuckSimulator {
    public static void main(String[] args) {
        simulateur DuckSimulator = nouveau DuckSimulator();
        RésuméDuckFactory duckFactory = new CountingDuckFactory();

        simulateur.simuler( canardFactory );
    }

    void simuler( RésuméDuckFactory duckFactory ) {
        Canard colvert cananard = duckFactory.createMallardDuck();
        cancanard roux cananard ... duckFactory.createRedheadDuck();
        Appel de canard cananant = duckFactory.createDuckCall();
        Canard en caoutchouc cananant = duckFactory.createRubberDuck();
        Oie cancanante = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator : avec fabrique abstraite ");

        simuler(canard colvert);
        simuler(redheadDuck);
        simuler(duckCall);
        simuler(rubberDuck);
        simuler(oieCanard);

        System.out.println("Les canards canancaient " +
            QuackCounter.getQuacks() + "
            fois");
    }

    void simuler(canard canaille) {
        canard.coin-coin();
    }
}

```

Nous créons d'abord l'usine que nous allons passer dans le simuler() méthode.

Le simuler()
la méthode prend un RésuméDuckFactory et l'utilise pour créer des canards plutôt que les instancier directement.

Rien ne change voilà ! Le même vieux code.

Voici le résultat en utilisant l'usine...

Comme la dernière fois, mais cette fois nous veillons à ce que les canards sont tous décorés parce que nous utilisons CountingDuckFactory.

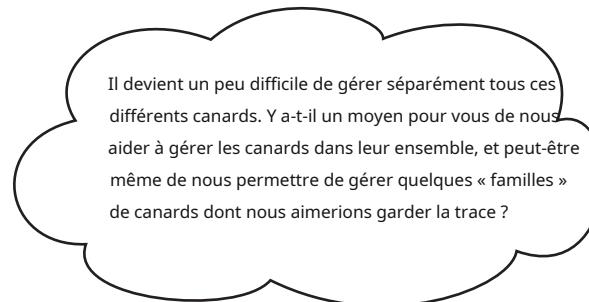


```
Fenêtre d'édition de fichier Aide EggFactory
% Java DuckSimulator
Simulateur de canard : avec Abstract Factory
Quack
Charlatan
Kwak
Grincer
Klaxonner
4 charlatans ont été comptés %
```



Sharpen your pencil

Nous instancions toujours directement Geese en nous appuyant sur des classes concrètes. Pouvez-vous écrire une fabrique abstraite pour Geese ? Comment devrait-elle gérer la création de « canards oies » ?



Ah, il veut gérer un troupeau de canards.

Voici une autre bonne question du Ranger Brewer : pourquoi gérons-nous les canards individuellement ?

Ce n'est pas très maniable!



```
Canard colvert cancanant = duckFactory.createMallardDuck(); Canard  
rouge cancanant = duckFactory.createRedheadDuck(); Appel de  
canard cancanant = duckFactory.createDuckCall(); Canard en  
caoutchouc cancanant = duckFactory.createRubberDuck(); Canard  
d'oie cancanant = new GooseAdapter(new Goose());
```

```
simuler(canard colvert);  
simuler(redheadDuck);  
simuler(duckCall);  
simuler(rubberDuck);  
simuler(oieCanard);
```

Ce dont nous avons besoin, c'est d'un moyen de parler de collections de canards et même de sous-collections de canards (pour traiter la demande de famille du Ranger Brewer). Il serait également intéressant de pouvoir appliquer des opérations sur l'ensemble des canards.

Quel modèle peut nous aider ?

⑫ Créons une volée de canards (en fait, une volée de Quackables).

Vous vous souvenez du modèle composite qui nous permet de traiter une collection d'objets de la même manière que des objets individuels ? Quel meilleur composite qu'un troupeau de charlatans !

Voyons comment cela va fonctionner :

```

la classe publique Flock implémente Quackable {
    Liste<Quackable> quackers = new ArrayList<Quackable>();

    public void add(quacker charlatan) {
        charlatans.add(charlatan);
    }

    public void charlatan() {
        Itérateur<Quackable> itérateur = quackers.iterator(); while
        (itérateur.hasNext()) {
            Quackable quacker = iterator.next();
            quacker.quack();
        }
    }
}

N'oubliez pas que le composite doit implémenter la
même interface que les éléments leaf. Nos éléments
leaf sont des Quackables.

Nous utilisons une ArrayList à l'intérieur de chaque Flock
pour contenir les Quackables qui appartiennent au Flock.

La méthode add() ajoute
un Quackable au Flock.

Passons maintenant à la méthode quack() — après tout, le Flock est également un
Quackable. La méthode quack() de Flock doit fonctionner sur l'ensemble du Flock. Ici,
nous parcourons l'ArrayList et appelons quack() sur chaque élément.

```



Le code de près

Avez-vous remarqué que nous avons essayé de vous faire découvrir un modèle de conception sans le mentionner ?

```

public void charlatan() {
    Itérateur<Quackable> itérateur = quackers.iterator(); while
    (itérateur.hasNext()) {
        Quackable quacker = iterator.next();
        quacker.quack();
    }
}

```

Et voilà ! Le modèle Iterator à l'œuvre !

(13) Nous devons maintenant modifier le simulateur.

Notre composite est prêt ; nous avons juste besoin d'un peu de code pour rassembler les canards dans la structure composite.

```
classe publique DuckSimulator {
```

```
// méthode principale ici
```

```
void simuler(AbstractDuckFactory duckFactory) {
    Canard rouge cancanant = duckFactory.createRedheadDuck(); Appel
    de canard cancanant = duckFactory.createDuckCall(); Canard en
    caoutchouc cancanant = duckFactory.createRubberDuck(); Canard
    d'oie cancanant = new GooseAdapter(new Goose());
```

Créez tous les Charlats, comme avant.

```
System.out.println("\nDuck Simulator : avec composite - Troupeaux");
```

```
Troupeau flockOfDucks = nouveau Troupeau();
```

Nous créons d'abord un troupeau et le chargeons de Quackables.

```
flockOfDucks.add(canardroux);
flockOfDucks.add(appeldecanard);
flockOfDucks.add(canarden caoutchouc);
flockOfDucks.add(canardoie);
```

Ensuite, nous créons un nouveau troupeau de colverts.

```
Troupeau flockOfMallards = nouveau troupeau() ;
```

Nous voici créer une petite famille de les colverts...

```
Canard colvert cancanantUn = duckFactory.createMallardDuck();
Canard colvert cancanantDeux = duckFactory.createMallardDuck();
Canard colvert cancanantTrois = duckFactory.createMallardDuck();
Canard colvert cancanantQuatre = duckFactory.createMallardDuck();
```

```
flockOfMallards.add(mallardOne);
flockOfMallards.add(mallardDeux);
flockOfMallards.add(mallardTrois);
flockOfMallards.add(mallardQuatre);
```

... et les ajouter au troupeau de colverts.

```
troupeauDeCanards.add(troupeauDeCanards colverts);
```

Ensuite, nous ajoutons le troupeau de colverts au troupeau principal.

```
System.out.println("\nDuck Simulator : Simulation d'un troupeau entier ");
simuler(troupeauDecanards);
```

Testons l'ensemble du Flock !

```
System.out.println("\nDuck Simulator : Simulation de troupeau de colverts ");
simuler(troupeauDeCanards colverts);
```

Alors testons simplement le troupeau de canards colverts.

```
System.out.println("\nLes canards cancaient " +
    QuackCounter.getQuacks() + "
    fois");
}
```

Enfin, donnons les données au charlatan.

```
void simuler(canard canaille) {
```

```
    canard.coin-coin();
}
```

Rien n'a besoin de changer ici : un troupeau est un charlatan !

Essayons-le...

```
Aide de la fenêtre d'édition de fichier FlockADuck
% Java DuckSimulator

Simulateur de canards : avec Composite - Flocks

Simulateur de canard : Simulation de cancan de troupeau
entier
Kwak Voici le premier troupeau.
Grincer

Klaxonner
Charlatan
Charlatan
Charlatan
Charlatan
Charlatan

Simulateur de canard : Simulation de troupeau de colverts
Charlatan Et maintenant les colverts.
Charlatan
Charlatan
Les canards ont cancané 11 fois

Les données semblent bien (rappelez-vous le l'oie ne comprend pas dénombré).
```

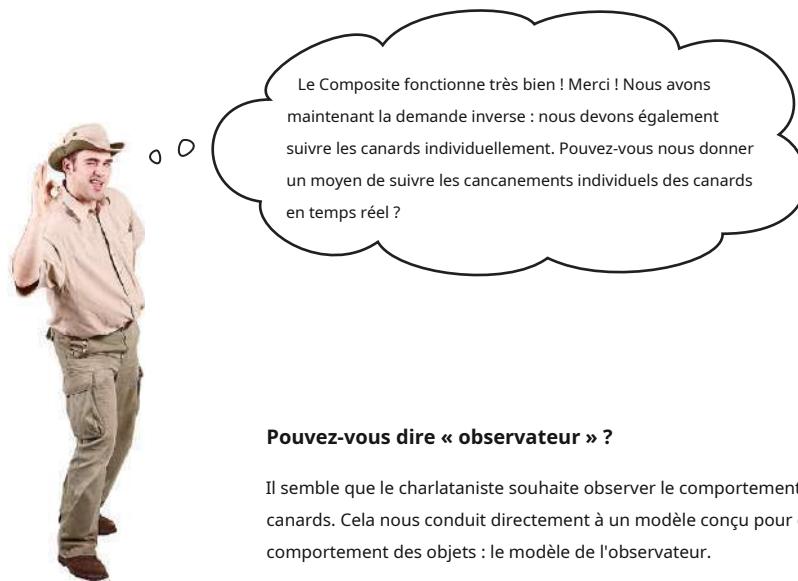


Sécurité versus transparence

Vous vous souviendrez peut-être que dans le chapitre sur les modèles composites, les composites (les menus) et les feuilles (les éléments de menu) avaient le même ensemble exact de méthodes, y compris la méthode `add()`. Comme ils avaient le même ensemble de méthodes, nous pouvions appeler des méthodes sur des `MenuItem`s qui n'avaient pas vraiment de sens (comme essayer d'ajouter quelque chose à un `MenuItem` en appelant `add()`). L'avantage de cela était que la distinction entre les feuilles et les composites était transparente : le client n'avait pas besoin de savoir s'il avait affaire à une feuille ou à un composite ; il appelaient simplement les mêmes méthodes sur les deux.

Ici, nous avons décidé de garder les méthodes de maintenance des enfants du composite séparées des nœuds feuilles : c'est-à-dire que seuls les troupeaux ont la méthode `add()`. Nous savons que cela n'a pas de sens d'essayer d'ajouter quelque chose à un canard, et dans cette implémentation, vous ne pouvez pas. Vous ne pouvez ajouter qu'à un troupeau. Cette conception est donc plus sûre — vous ne pouvez pas appeler des méthodes qui n'ont pas de sens sur les composants — mais c'est moins transparent. Le client doit maintenant savoir qu'un `Quackable` est un `Flock` pour pouvoir lui ajouter des `Quackables`.

Comme toujours, il y a des compromis lorsque vous faites de la conception OO et vous devez les prendre en compte lorsque vous créez vos propres composites.



Pouvez-vous dire « observateur » ?

Il semble que le charlataniste souhaite observer le comportement individuel des canards. Cela nous conduit directement à un modèle conçu pour observer le comportement des objets : le modèle de l'observateur.

14

Nous avons d'abord besoin d'une interface pour notre sujet.

N'oubliez pas que le sujet est l'objet observé. Appelons-le quelque chose de plus mémorable : pourquoi pas Observable ? Un Observable a besoin de méthodes pour enregistrer et notifier les observateurs. Nous pourrions également avoir une méthode pour supprimer les observateurs, mais nous garderons ici une implémentation simple et laisserons cela de côté.

QuackObservable est l'interface que les Quackables doivent implémenter s'ils veulent être observés.

```
interface publique QuackObservable {
    public void registerObserver(Observer observer); public void
    notifyObservers();
}
```

Il dispose également d'une méthode de notification des observateurs.

Il dispose d'une méthode pour enregistrer les observateurs. Tout objet implémentant l'interface Observer peut écouter les charlatans. Nous définirons l'interface Observer dans une seconde.

Nous devons maintenant nous assurer que tous les Quackables implémentent cette interface...

```
interface publique Quackable     étend QuackObservable {
    public void charlatan();
}
```

Nous étendons donc l'interface Quackable avec QuackObserver.

(15)

Maintenant, nous devons nous assurer que toutes les classes concrètes qui implémentent Quackable peuvent gérer le fait d'être un QuackObservable.

Nous pourrions aborder cela en implémentant l'enregistrement et la notification dans chaque classe (comme nous l'avons fait au chapitre 2). Mais nous allons le faire un peu différemment cette fois-ci : nous allons encapsuler le code d'enregistrement et de notification dans une autre classe, l'appeler Observable et le composer avec QuackObservable. De cette façon, nous n'écrivons le vrai code qu'une seule fois et QuackObservable n'a besoin que de suffisamment de code pour déléguer à la classe d'assistance Observable.

Commençons par la classe d'aide Observable.



QuackObservable

Observable implémente toutes les fonctionnalités dont un Quackable a besoin pour être un observable. Il nous suffit de le connecter à une classe et de faire en sorte que cette classe délègue à Observable.

Observable doit implémenter QuackObservable car ce sont les mêmes appels de méthode qui vont lui être délégués.



```
la classe publique Observable implémente QuackObservable {
    Liste<Observateur> observers = new ArrayList<Observateur>();
    QuackObservable canard;
```

```
public Observable(QuackObservable canard) {
    ce.canard = canard;
}
```

```
public void registerObserver(Observateur observateur) {
    observateurs.add(observateur);
}
```

Dans le constructeur, nous recevons le QuackObservable qui utilise cet objet pour gérer son comportement observable. Consultez la méthode notifyObservers() ci-dessous ; vous verrez que lorsqu'une notification se produit, Observable transmet cet objet afin que l'observateur sache quel objet cancané.

Voici le code pour enregistrer un observateur.

```
public void notifierObservers() {
    Itérateur itérateur = observers.iterator(); while
    (itérateur.hasNext()) {
        Observateur observer = itérateur.next();
        observer.update(canard);
    }
}
```



Et le code pour faire les notifications.

Voyons maintenant comment une classe Quackable utilise cette aide...

16 Intégrez l'assistant Observable avec les classes Quackable.

Cela ne devrait pas être trop grave. Il suffit de s'assurer que les classes Quackable sont composées d'un Observable et qu'elles savent comment lui déléguer des tâches. Après cela, elles sont prêtes à être des Observables. Voici l'implémentation de MallardDuck ; les autres canards sont les mêmes.

la classe publique MallardDuck implémente Quackable {

Observable observable;

Chaque Quackable possède une variable d'instance Observable.

public MallardDuck() {

observable = nouveau Observable(ceci);

}



Dans le constructeur, nous créons un Observable et lui passons une référence à l'objet MallardDuck.

public void charlatan() {

System.out.println("Coin-coin");

notifierObservers();

}



Lorsque nous cançons, nous devons le faire savoir aux observateurs.

public void registerObserver(Observateur observateur) {

observable.registerObserver(observateur);

}

public void notifierObservers() {

observable.notifierObservers();

}

}



Voici nos deux méthodes QuackObservable. Notez que nous délégons simplement à l'assistant.



Sharpen your pencil

Nous n'avons pas modifié l'implémentation d'un Quackable, le décorateur QuackCounter. Nous devons également en faire un Observable. Pourquoi n'écrivez-vous pas celui-là :

(17)

Nous y sommes presque ! Il ne nous reste plus qu'à travailler sur le côté observateur du modèle.

Nous avons implémenté tout ce dont nous avons besoin pour les Observables ; maintenant, nous avons besoin de quelques Observateurs. Nous commencerons par l'interface Observer :



L'interface Observer n'a qu'une seule méthode, update(), à laquelle est transmis le QuackObservable qui cancane.

```
interface publique Observer {
    public void update(canard QuackObservable);
}
```

Maintenant, nous avons besoin d'un observateur : où sont ces charlatans ?!

Nous devons implémenter l'interface Observer sinon nous ne pourrons pas nous inscrire auprès d'un QuackObservable.



```
classe publique Quackologist implémente Observer {

    public void update(QuackObservable canard) {
        System.out.println("Quackologist: " + duck + " vient de cancaner.");
    }
}
```



Le Quackologist est simple ; il n'a qu'une seule méthode, update(), qui imprime le Quackable qui vient de cancaner.



Sharpen your pencil

Et si un charlataniste veut observer un troupeau entier ? Qu'est-ce que cela signifie ? Pensez-y comme ceci : si nous observons un composite, alors nous observons tout ce qui se trouve dans le composite. Ainsi, lorsque vous vous enregistrez dans un troupeau, le composite du troupeau s'assure que vous êtes enregistré avec tous ses enfants (désolé, tous ses petits charlatans), ce qui peut inclure d'autres troupeaux.

Allez-y et écrivez le code de l'observateur Flock avant d'aller plus loin.

- 18 Nous sommes prêts à observer. Mettons à jour le simulateur et essayons :

```

classe publique DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulateur = nouveau DuckSimulator();
        AbstractDuckFactory duckFactory = nouveau CountingDuckFactory();

        simulateur.simulate(duckFactory);
    }

    void simuler(AbstractDuckFactory duckFactory) {

        // créer des usines à canards et des canards ici

        // créer des troupeaux ici

        System.out.println("\nDuck Simulator : avec observateur ");

        Quackologist quackologist = new Quackologist();
        flockOfDucks.registerObserver(quackologist);
    }

    simuler(troupeauDeCanards);

    System.out.println("\nLes canards cancanaien " +
        QuackCounter.getQuacks() + "
        fois");

}

```

```

void simuler(canard canaille) {
    canard.coin-coin();
}

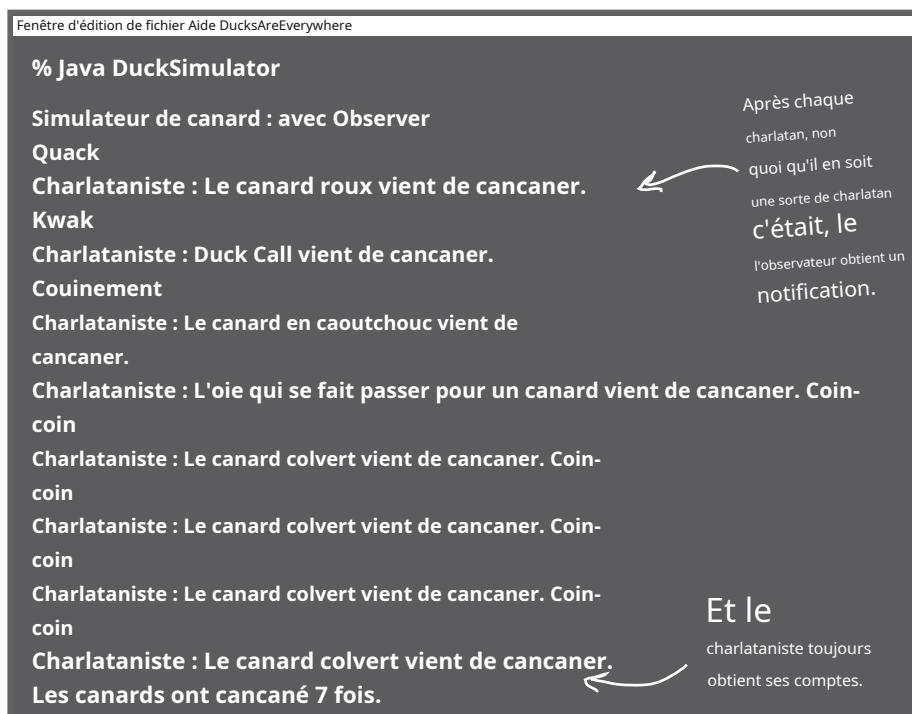
```

Tout ce que nous faisons ici est de créer un charlatan et de le définir comme observateur du troupeau.

Cette fois nous allons nous simulons simplement l'ensemble du troupeau.

Essayons et voyons comment cela fonctionne !

C'est la grande finale. Cinq, non, six, modèles ont été réunis pour créer cet incroyable simulateur de canard. Sans plus attendre, nous vous présentons DuckSimulator !



Q: Donc c'était un motif composé ?
UN: Non il s'agissait simplement d'un ensemble de modèles fonctionnant ensemble. Un modèle composé est un ensemble de quelques modèles qui sont combinés pour résoudre un problème général. Nous sommes sur le point d'examiner le modèle composé Modèle-Vue-Contrôleur ; il s'agit d'une collection de quelques modèles qui a été utilisée à maintes reprises dans de nombreuses solutions de conception.

Q: La véritable beauté des modèles de conception est que je peux prendre un problème et commencer à lui appliquer des modèles jusqu'à ce que j'aie une solution. N'est-ce pas ?
UN: Paul. Nous avons fait cet exercice avec les canards pour vous montrer comment les motifs peuvent travailler ensemble. Vous ne voudriez jamais aborder une conception comme nous venons de le faire. En fait, il peut y avoir des solutions pour certaines parties du simulateur de canard pour lesquelles certains de ces modèles étaient exagérés. Parfois, le simple fait d'utiliser de bons principes de conception OO peut résoudre un problème à lui seul.

Nous en parlerons plus en détail dans le prochain chapitre, mais vous ne devez appliquer des modèles que lorsque et où ils ont du sens. Vous ne devez jamais commencer avec l'intention d'utiliser des modèles juste pour le plaisir. Vous devez considérer la conception du simulateur de canard comme forcée et artificielle. Mais bon, c'était amusant et cela nous a donné une bonne idée de la façon dont plusieurs modèles peuvent s'intégrer dans une solution.

Qu'avons-nous fait ?

Nous avons commencé avec un groupe de Quackables...

Une oie est arrivée et a voulu agir comme un Quackable aussi. Nous avons donc utilisé le *Modèle d'adaptateur* pour adapter l'oie à un Quackable. Maintenant, vous pouvez appeler `quack()` sur une oie enveloppée dans l'adaptateur et elle klaxonnera !

Ensuite, les charlatans ont décidé qu'ils voulaient compter les charlatans. Nous avons donc utilisé le *Modèle de décorateur* pour ajouter un décorateur `QuackCounter` qui garde une trace du nombre de fois que `quack()` est appelé, puis délègue le `quack` au Quackable qu'il enveloppe.

Mais les Quackologistes craignaient d'oublier d'ajouter le décorateur `QuackCounter`. Nous avons donc utilisé le *Modèle d'usine abstraite* pour leur créer des canards. Désormais, chaque fois qu'ils veulent un canard, ils en demandent un à l'usine, et celle-ci leur en renvoie un décoré. (Et n'oubliez pas, ils peuvent aussi faire appel à une autre usine de canards s'ils veulent un canard non décoré !)

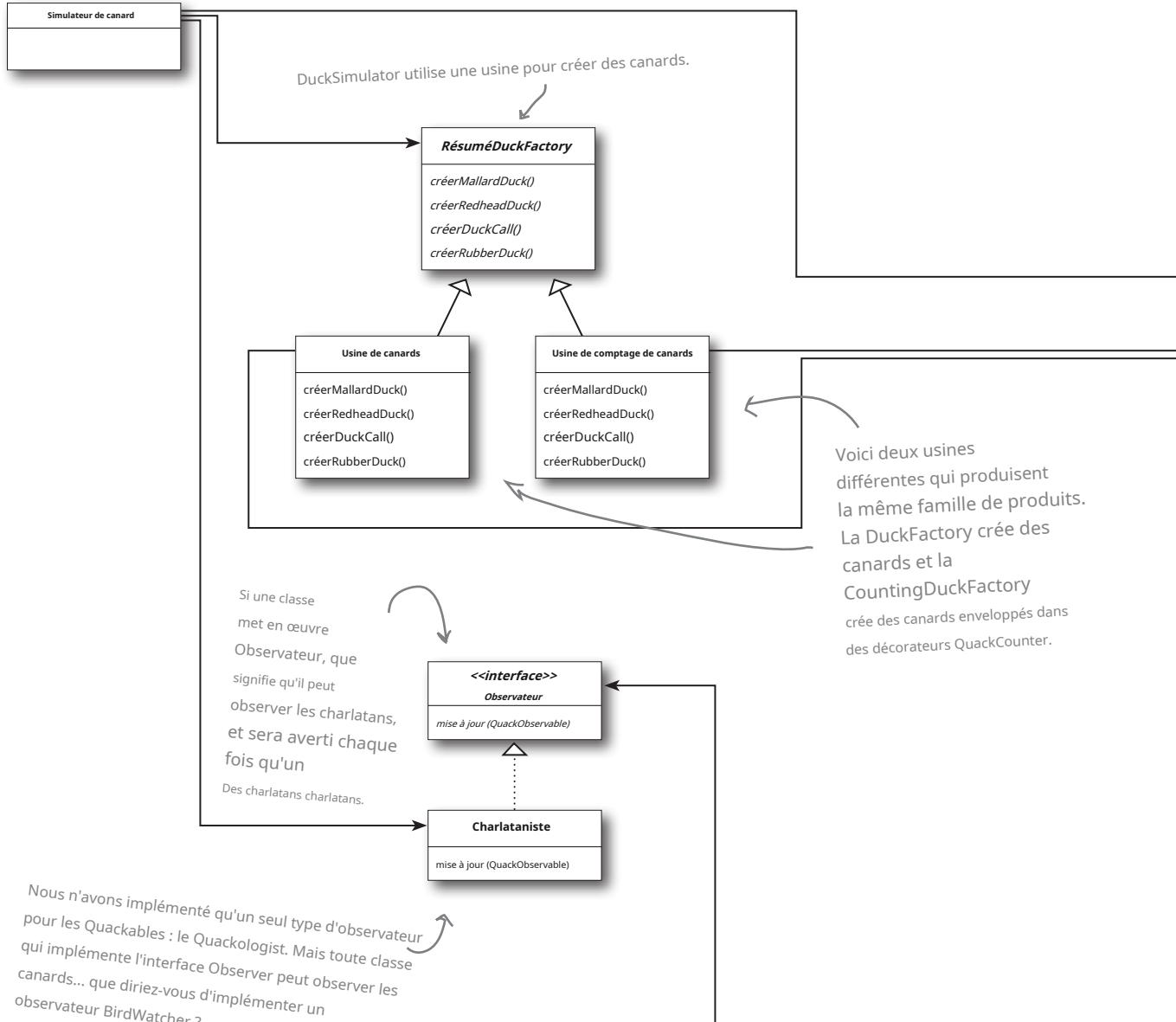
Nous avons eu des problèmes de gestion pour garder la trace de tous ces canards, oies et charlatans. Nous avons donc utilisé le *Modèle composite* pour regrouper les Quackables en groupes. Le modèle permet également au Quackologist de créer des sous-groupes pour gérer les familles de canards. Nous avons utilisé le *Modèle d'itérateur* dans notre implémentation en utilisant l'itérateur de `java.util` dans `ArrayList`.

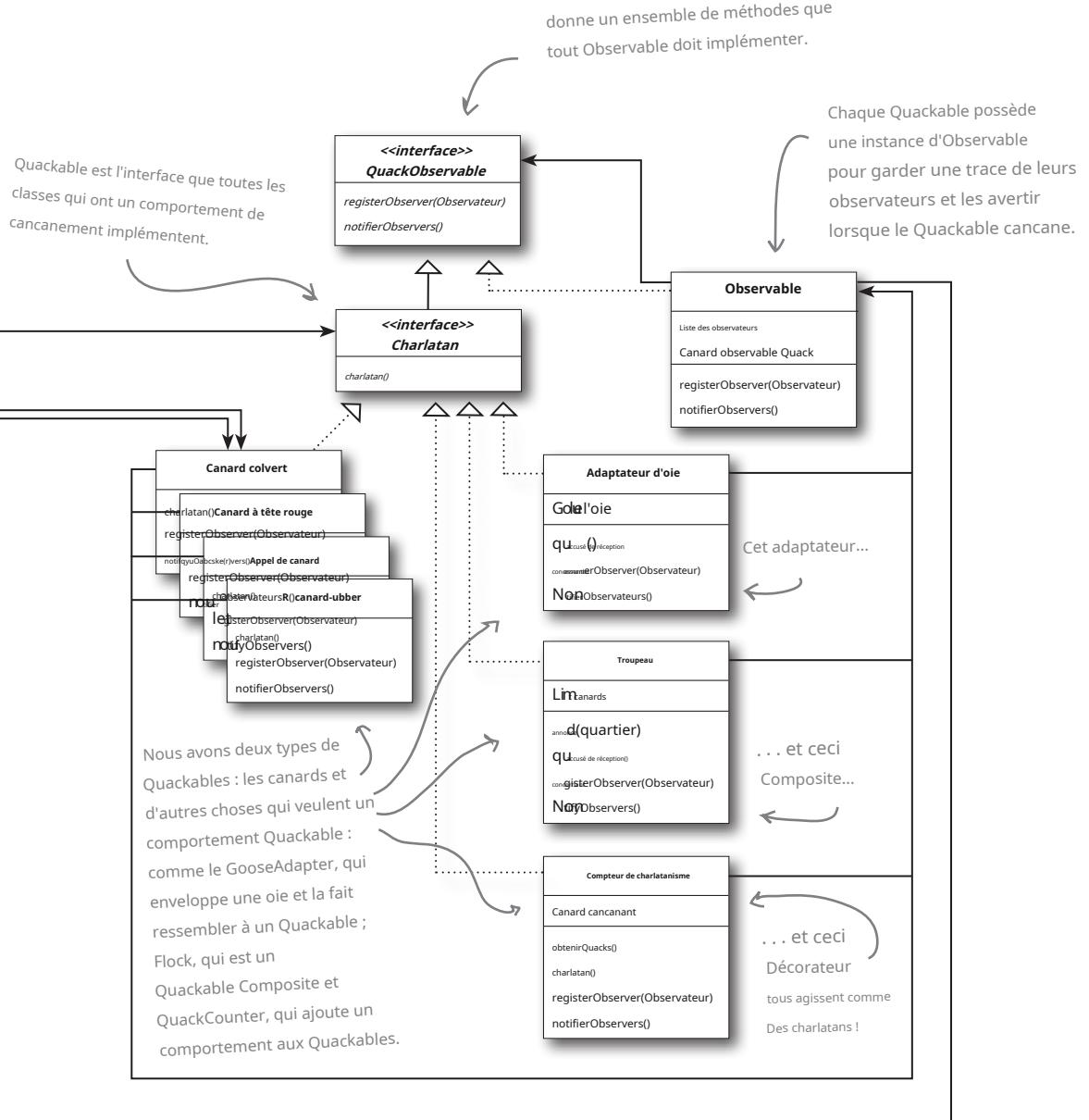
Les charlatans voulaient également être avertis lorsqu'un charlatan faisait cancanement. Nous avons donc utilisé le *Modèle d'observateur* pour permettre aux Quackologists de s'enregistrer comme Observateurs Quackables. Ils sont désormais avertis à chaque fois qu'un Quackable cancane. Nous avons à nouveau utilisé l'itérateur dans cette implémentation. Les Quackologists peuvent même utiliser le modèle Observateur avec leurs composites.



~~Une vue d'ensemble : le diagramme de classes~~

Nous avons regroupé de nombreux modèles dans un petit simulateur de canard ! Voici le résumé de ce que nous avons fait :





Le roi des motifs composés

Si Elvis était un motif composé, son nom serait Modèle-Vue-Contrôleur, et il chanterait une petite chanson comme celle-ci...

Modèle, vue, contrôleur

Paroles et musique de James Dempsey.

MVC est un paradigme pour factoriser votre code en segments fonctionnels, pour que votre cerveau n'explose pas. Pour parvenir à la réutilisabilité, vous devez garder ces limites propres

Modèle d'un côté, vue de l'autre, le contrôleur est entre les deux.



Vue du modèle, il comporte trois couches comme les Oreos et le contrôleur de vue du modèle

Vue du modèle, vue du modèle, contrôleur de vue du modèle

Les objets de modèle représentent la raison d'être de votre application. Objets personnalisés contenant des données, de la logique, etc. Vous créez des classes personnalisées, dans le domaine de problème de votre application, vous pouvez choisir de les réutiliser avec toutes les vues, mais les objets de modèle restent les mêmes.

Vous pouvez modéliser un accélérateur et un collecteur.

Modéliser le pas d'un enfant de deux ans.

Modélisez une bouteille de bon Chardonnay Modélisez tous les coups de glotte que les gens disent Modélisez le fait de dorloter des œufs à la coque Vous pouvez modéliser le dandinement des jambes d'Hexley

Vue du modèle, vous pouvez modéliser tous les modèles qui posent pour le contrôleur de vue du modèle GQ

Java aussi !
Les objets de vue ont tendance à être des contrôles utilisés pour afficher et éditer. Cocoa en a beaucoup, bien écrits à son actif. Prenez un NSTextView, donnez-lui n'importe quelle vieille chaîne Unicode L'utilisateur peut interagir avec lui, il peut contenir presque tout, mais la vue ne connaît pas le modèle
Cette chaîne pourrait être un numéro de téléphone ou les œuvres d'Aristote
Gardez le couplage lâche et ainsi atteindre un niveau massif de réutilisation

Vue du modèle, le tout très joliment rendu en bleu aqua
Contrôleur de vue du modèle

Vous vous demandez probablement maintenant Vous vous demandez probablement comment les données circulent entre le modèle et la vue Le contrôleur doit servir d'intermédiaire entre les changements d'état de chaque couche Pour synchroniser les données des deux Il extrait et pousse chaque valeur modifiée

Vue du modèle, félicitations à l'équipe de Smalltalk !
Contrôleur de vue du modèle

Vue du modèle, elle se prononce Oh Oh et non Ooo Ooo
Contrôleur de vue du modèle

Il reste encore un peu de cette histoire.
Encore quelques kilomètres sur cette route.
Personne ne semble tirer beaucoup de gloire de l'écriture du code du contrôleur.

Eh bien, la mission du modèle est cruciale et la vue est magnifique
Je suis peut-être paresseux, mais parfois c'est juste fou. La quantité de code que j'écris n'est que de la colle.
Et ce ne serait pas si tragique, mais le code ne fait pas de magie. Il ne fait que déplacer des valeurs.

Et je ne veux pas être méchant,
mais ça devient répétitif.
Faire tout ce que font les contrôleurs

Et j'aimerais avoir une pièce de dix cents à chaque fois
J'ai envoyé une `StringValue` `TextField`.

Comment allons-nous retirer toute cette colle
Modèle Vue Contrôleur

Les contrôleurs connaissent très intimement le modèle et la vue
Ils utilisent souvent un codage en dur qui peut être préoccupant en termes de réutilisabilité.
Mais vous pouvez désormais connecter chaque clé de modèle que vous sélectionnez à n'importe quelle propriété de vue

Et une fois que vous commencez à lier
Je pense que vous trouverez moins de code dans votre arborescence de sources

Ouais, je sais que j'étais ravi par les choses qu'ils ont automatisées et les choses que vous obtenez gratuitement
Et je pense qu'il vaut la peine de répéter tout le code dont vous n'aurez pas besoin lorsque vous le connecterez à IB.

La vue du modèle gère également plusieurs sélections
Contrôleur de vue du modèle

Modèle Vue, je parie que j'expédie mon application avant vous
Modèle Vue Contrôleur

Vue du modèle



Oreille pouvoir

Ne vous contentez pas de lire ! Après tout, il s'agit d'un livre à lire en premier... consultez cette URL :

<https://www.youtube.com/watch?v=YYvOGPMLVDo>

Asseyez-vous et écoutez-le.

Jolie chanson, mais est-ce que c'est vraiment censé m'apprendre ce qu'est le modèle-vue-contrôleur ? J'ai déjà essayé d'apprendre le MVC et ça m'a fait mal au cerveau.



Les modèles de conception sont la clé pour comprendre MVC.

Nous essayions simplement de vous mettre en appétit avec cette chanson. Je vous dis quoi, après avoir fini de lire ce chapitre, revenez en arrière et réécoutez la chanson : vous vous amuserez encore plus.

Il semble que vous ayez déjà eu une mauvaise expérience avec MVC ? C'est le cas de la plupart d'entre nous. Vous avez probablement entendu d'autres développeurs vous dire que cela a changé leur vie et pourrait éventuellement créer la paix dans le monde. C'est un modèle composé puissant, c'est sûr, et même si nous ne pouvons pas affirmer qu'il créera la paix dans le monde, il vous fera économiser des heures d'écriture de code une fois que vous le connaîtrez.

Mais il faut d'abord l'apprendre, n'est-ce pas ? Eh bien, il y aura une grande différence cette fois-ci, car *maintenant vous connaissez les motifs !*

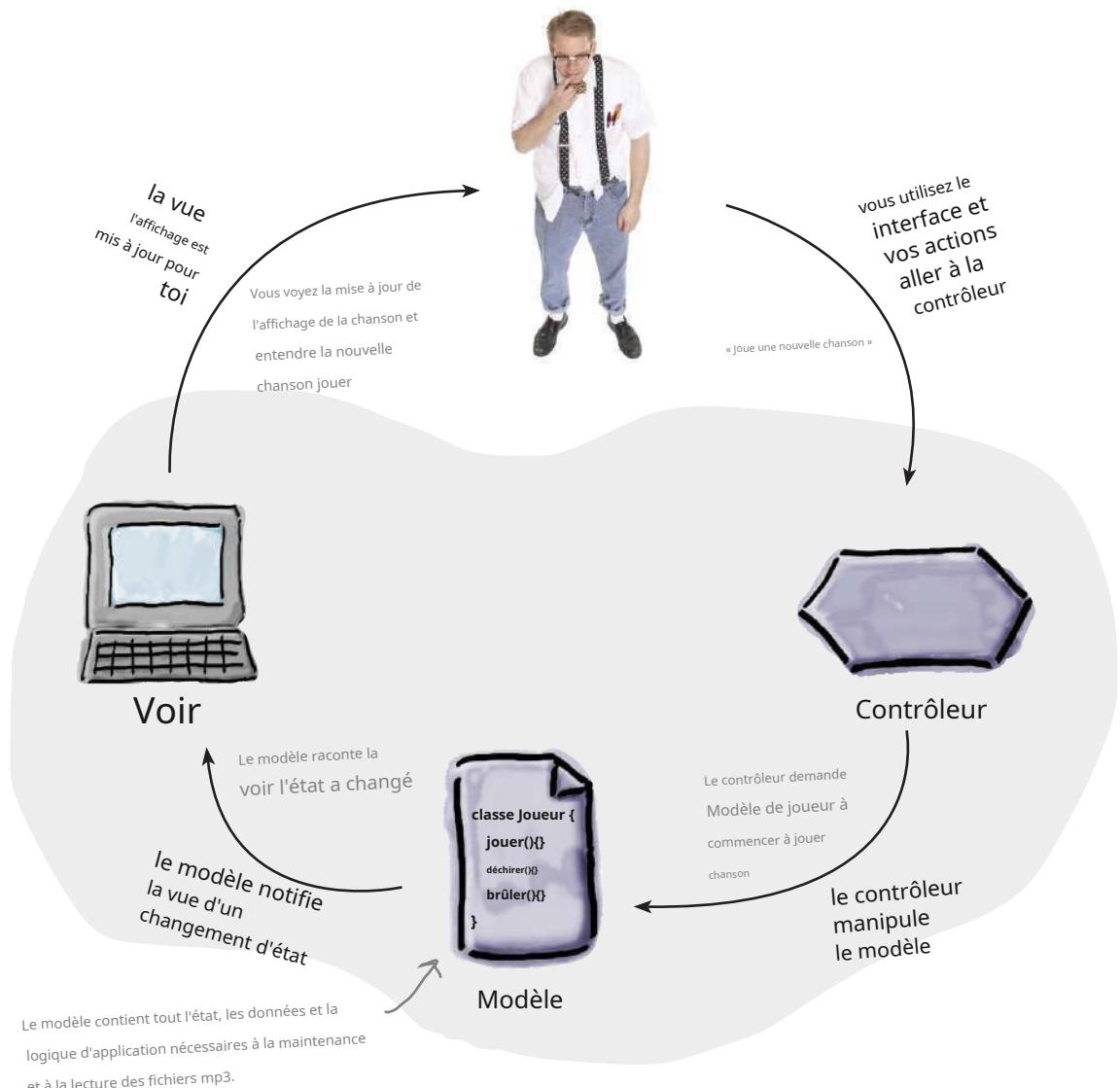
C'est vrai, les modèles sont la clé du MVC. Apprendre le MVC de haut en bas est difficile ; peu de développeurs y parviennent. Voici le secret pour apprendre le MVC : *ce ne sont que quelques modèles assemblés.* Lorsque vous abordez l'apprentissage du MVC en examinant les modèles, tout d'un coup, cela commence à avoir du sens.

C'est parti. Cette fois-ci, vous allez réussir le MVC !

Découvrez le Modèle-Vue-Contrôleur

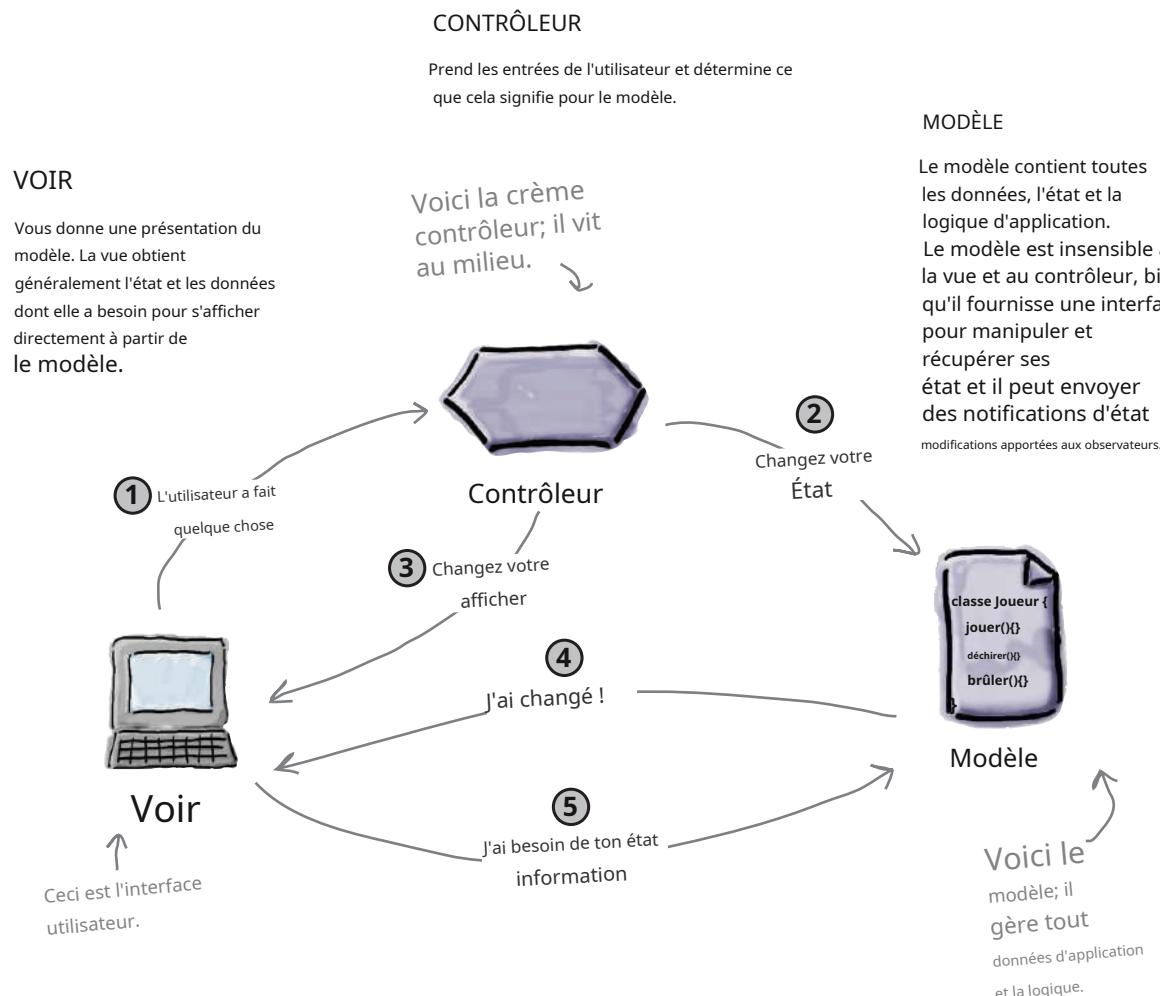
Imaginez que vous utilisez votre lecteur de musique préféré, comme iTunes. Vous pouvez utiliser son interface pour ajouter de nouvelles chansons, gérer des listes de lecture et renommer des pistes. Le lecteur se charge de maintenir une petite base de données de toutes vos chansons ainsi que de leurs noms et données associés. Il se charge également de lire les chansons et, ce faisant, l'interface utilisateur est constamment mise à jour avec le titre de la chanson en cours, la durée d'exécution, etc.

Et bien, en dessous de tout cela se trouve le Modèle-Vue-Contrôleur...



Un regard plus attentif...

La description du lecteur de musique nous donne une vue d'ensemble de MVC, mais elle ne vous aide pas vraiment à comprendre le fonctionnement du modèle composé, la manière dont vous en créeriez un vous-même ou pourquoi c'est une si bonne chose. Commençons par parcourir les relations entre le modèle, la vue et le contrôleur, puis nous examinerons à nouveau le sujet du point de vue des modèles de conception.



1 Vous êtes l'utilisateur : vous interagissez avec la vue.

La vue est votre fenêtre sur le modèle. Lorsque vous effectuez une action sur la vue (comme cliquer sur le bouton Lecture), la vue indique au contrôleur ce que vous avez fait. C'est le rôle du contrôleur de gérer cela.

2 Le contrôleur demande au modèle de changer son état.

Le contrôleur prend en compte vos actions et les interprète. Si vous cliquez sur un bouton, c'est le rôle du contrôleur de déterminer ce que cela signifie et comment le modèle doit être manipulé en fonction de cette action.

3 Le contrôleur peut également demander que la vue change.

Lorsque le contrôleur reçoit une action de la vue, il peut être amené à indiquer à la vue de changer en conséquence. Par exemple, le contrôleur peut activer ou désactiver certains boutons ou éléments de menu dans l'interface.

4 Le modèle notifie la vue lorsque son état a changé.

Lorsqu'un élément change dans le modèle, en fonction d'une action que vous avez effectuée (comme cliquer sur un bouton) ou d'un autre changement interne (comme le démarrage de la chanson suivante de la liste de lecture), le modèle notifie à la vue que son état a changé.

5 La vue demande l'état au modèle.

La vue obtient l'état qu'elle affiche directement à partir du modèle. Par exemple, lorsque le modèle informe la vue qu'une nouvelle chanson a commencé à jouer, la vue demande le nom de la chanson au modèle et l'affiche. La vue peut également demander au modèle l'état à la suite d'une demande de modification de la vue par le contrôleur.

*there are no
Dumb Questions*

Q: Le contrôleur devient-il parfois un observateur du modèle ?

UN: Bien sûr. Dans certaines conceptions, le contrôleur s'enregistre auprès du modèle et est informé des modifications. Cela peut être le cas lorsqu'un élément du modèle affecte directement les contrôles de l'interface utilisateur. Par exemple, certains états du modèle peuvent imposer l'activation ou la désactivation de certains éléments de l'interface. Si tel est le cas, il incombe en réalité au contrôleur de demander à la vue de mettre à jour son affichage en conséquence.

Q: Tout ce que fait le contrôleur, c'est de prendre les entrées de l'utilisateur dans la vue et de les envoyer au modèle, n'est-ce pas ? Pourquoi l'avoir si c'est tout ce qu'il fait ? Pourquoi ne pas simplement avoir le code dans la vue elle-même ? Dans la plupart des cas, le contrôleur n'appelle-t-il pas simplement une méthode sur le modèle ?

UN: Le contrôleur fait plus que simplement « l'envoyer au modèle » ; il est responsable de interpréter l'entrée et manipuler le modèle en fonction de cette entrée. Mais votre vraie question est probablement : « Pourquoi ne puis-je pas simplement faire cela dans le code de la vue ? »

Vous pourriez le faire, mais vous ne le souhaitez pas pour deux raisons. Tout d'abord, vous compliquerez votre code de vue car il a désormais deux responsabilités : gérer l'interface utilisateur et gérer la logique de contrôle du modèle.

Ensuite, vous couplez étroitement votre vue au modèle. Si vous souhaitez réutiliser la vue avec un autre modèle, oubliez-le. Le contrôleur sépare la logique de contrôle de la vue et découpe la vue du modèle.

En gardant la vue et le contrôleur faiblement couplés, vous créez une conception plus flexible et extensible, qui peut plus facilement s'adapter aux changements ultérieurs.

Comprendre MVC comme un ensemble de modèles

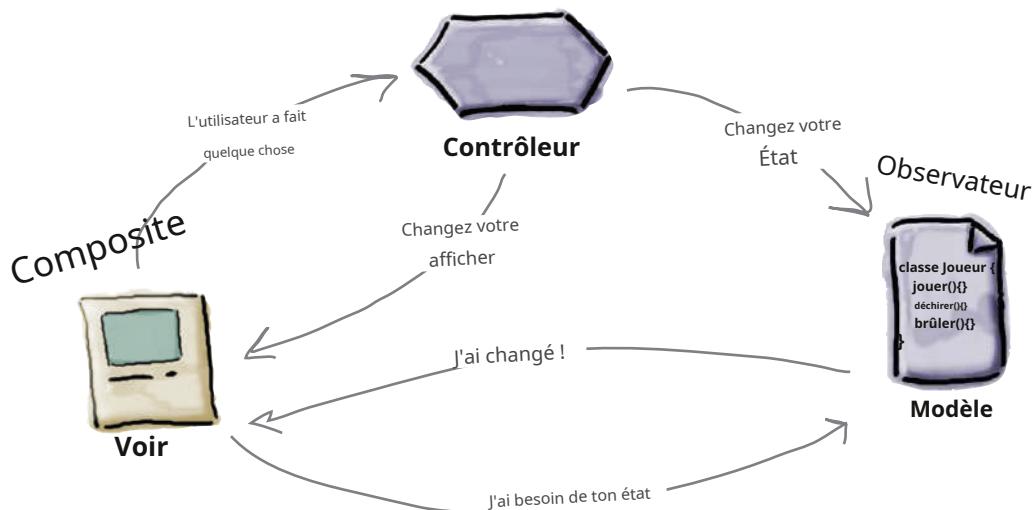
Nous avons déjà suggéré que le meilleur chemin pour apprendre MVC est de le voir pour ce qu'il est : un ensemble de modèles fonctionnant ensemble dans la même conception.

Commençons par le modèle : le modèle utilise Observer pour maintenir les vues et les contrôleur à jour sur les derniers changements d'état. La vue et le contrôleur, d'autre part, implémentent le modèle de stratégie. Le contrôleur est la stratégie de la vue, et il peut être facilement échangé avec un autre contrôleur si vous souhaitez un comportement différent. La vue elle-même utilise également un modèle en interne pour gérer les fenêtres, les boutons et les autres composants de l'affichage : le modèle composite.

Regardons cela de plus près :

Stratégie

La vue et le contrôleur implémentent le modèle de stratégie classique : la vue est un objet configuré avec une stratégie. Le contrôleur fournit la stratégie. La vue ne s'occupe que des aspects visuels de l'application et délègue au contrôleur toutes les décisions concernant le comportement de l'interface. L'utilisation du modèle de stratégie permet également de découpler la vue du modèle, car c'est le contrôleur qui est chargé d'interagir avec le modèle pour exécuter les demandes des utilisateurs. La vue ne sait rien de la manière dont cela se fait.



L'affichage est constitué d'un ensemble imbriqué de fenêtres, de panneaux, de boutons, d'étiquettes de texte, etc. Chaque composant d'affichage est un composite (comme une fenêtre) ou une feuille (comme un bouton). Lorsque le contrôleur demande à la vue de se mettre à jour, il n'a qu'à le dire au composant de la vue supérieure, et Composite s'occupe du reste.

Le modèle implémente le modèle Observer pour maintenir les objets intéressés à jour lorsque des changements d'état se produisent. L'utilisation du modèle Observer permet de garder le modèle complètement indépendant des vues et des contrôleurs. Il nous permet d'utiliser différentes vues avec le même modèle, voire d'utiliser plusieurs vues à la fois.

Observateur

Observable



Modèle

Mon état a modifié!

Observateurs



Voir

Tous ces observateurs seront notifiés chaque fois que l'état du modèle change.



Voir

Je voudrais m'inscrire en tant qu'observateur



Voir

Tout objet qui est intéressé par l'état les changements dans les registres du modèle avec le modèle en tant qu'observateur.

Le modèle n'a aucune dépendance vis-à-vis des spectateurs ou des contrôleurs !

Stratégie

La vue délégués à le contrôleur pour gérer le actions de l'utilisateur.



Voir

L'utilisateur a fait quelque chose



Contrôleur

Le contrôleur est la stratégie pour le vue — c'est l'objet qui sait comment gérer les actions de l'utilisateur.



Contrôleur

Nous pouvons échanger un autre comportement pour la vue en changeant le contrôleur.

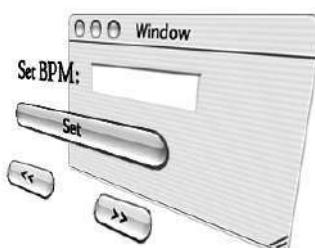
La vue se préoccupe uniquement de la présentation. Le contrôleur se préoccupe de traduire les entrées de l'utilisateur en actions sur le modèle.

Composite



Voir

peinture()



La vue est un composite de composants d'interface utilisateur graphique (étiquettes, boutons, saisie de texte, etc.). Le composant de niveau supérieur contient d'autres composants, qui contiennent d'autres composants, et ainsi de suite jusqu'à ce que vous arriviez aux nœuds feuilles.

Utiliser MVC pour contrôler le rythme...

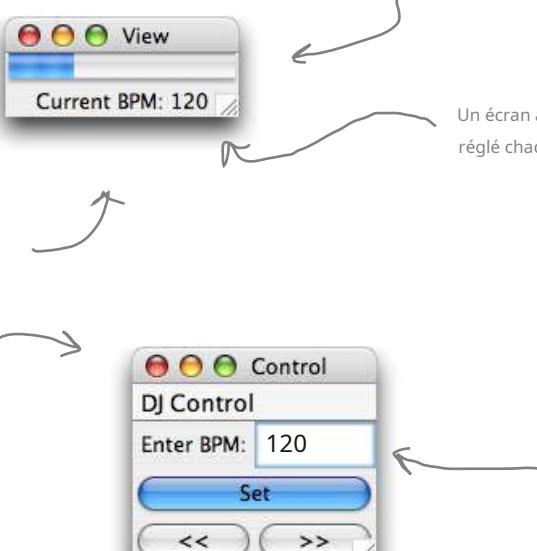
C'est votre tour d'être le DJ. Quand vous êtes DJ, tout est une question de rythme. Vous pouvez commencer votre mix avec un groove lent et lent à 95 battements par minute (BPM), puis faire monter la foule à 140 BPM de techno trance frénétique. Vous terminerez votre set avec un mix ambient doux à 80 BPM.

Comment vas-tu faire ça ? Tu dois contrôler le rythme et tu vas construire l'outil pour y arriver.



Découvrez le DJ Java

Commençons par **le voir** de l'outil. La vue vous permet de créer un rythme de batterie entraînant et d'ajuster ses battements par minute...



The screenshot shows two windows side-by-side. The left window is titled "View" and displays the text "Current BPM: 120". The right window is titled "Control" and contains a form with "Enter BPM: 120" and a "Set" button, along with increment/decrement arrows. A callout points to the "View" window with the text: "La vue comporte deux parties, la partie pour visualiser le état du modèle et de la partie permettant de contrôler les choses." Another callout points to the "Control" window with the text: "Vous pouvez entrer un BPM spécifique et cliquer sur le bouton Définir pour définir un nombre de battements par minute spécifique, ou vous pouvez utiliser les boutons d'augmentation et de diminution pour un réglage précis." A third callout points to the "View" window with the text: "Une barre pulsée indique le rythme en temps réel." A fourth callout points to the "Control" window with the text: "Un écran affiche les BPM actuels et est automatiquement réglé chaque fois que le BPM change." A fifth callout points to the increment arrow with the text: "Augmente le BPM par un battement par minute." A sixth callout points to the decrement arrow with the text: "Diminutions le BPM par un battement par minute."

Une barre pulsée indique le rythme en temps réel.

Un écran affiche les BPM actuels et est automatiquement réglé chaque fois que le BPM change.

La vue comporte deux parties, la partie pour visualiser le état du modèle et de la partie permettant de contrôler les choses.

Vous pouvez entrer un BPM spécifique et cliquer sur le bouton Définir pour définir un nombre de battements par minute spécifique, ou vous pouvez utiliser les boutons d'augmentation et de diminution pour un réglage précis.

Diminutions le BPM par un battement par minute.

Augmente le BPM par un battement par minute.

Voici quelques autres façons de contrôler la vue DJ...



Vous pouvez commencer le rythme en frappant choisir le départ élément de menu dans le menu « DJ Control ».

Avis Stop est désactivé jusqu'à ce que vous commence le rythme.

Vous utilisez le bouton Stop pour arrêter en rythme génération.



Avis de départ est désactivé après le rythme a commencé.

Toutes les actions de l'utilisateur sont envoyées au contrôleur.

Le contrôleur est au milieu...

Le **contrôleur** se situe entre la vue et le modèle. Il prend en compte votre saisie, comme la sélection de Démarrer dans le menu DJ Control, et la transforme en une action sur le modèle pour démarrer la génération de rythmes.

Le contrôleur prend les informations de l'utilisateur et détermine comment les traduire en requêtes sur le modèle.



Contrôleur

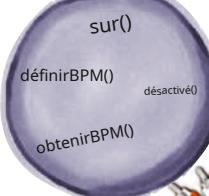
N'oublions pas le modèle qui se cache derrière tout cela...

Vous ne pouvez pas voir le **modèle**, mais on peut l'entendre. Le modèle se trouve en dessous de tout le reste, gérant le rythme et pilotant les haut-parleurs.

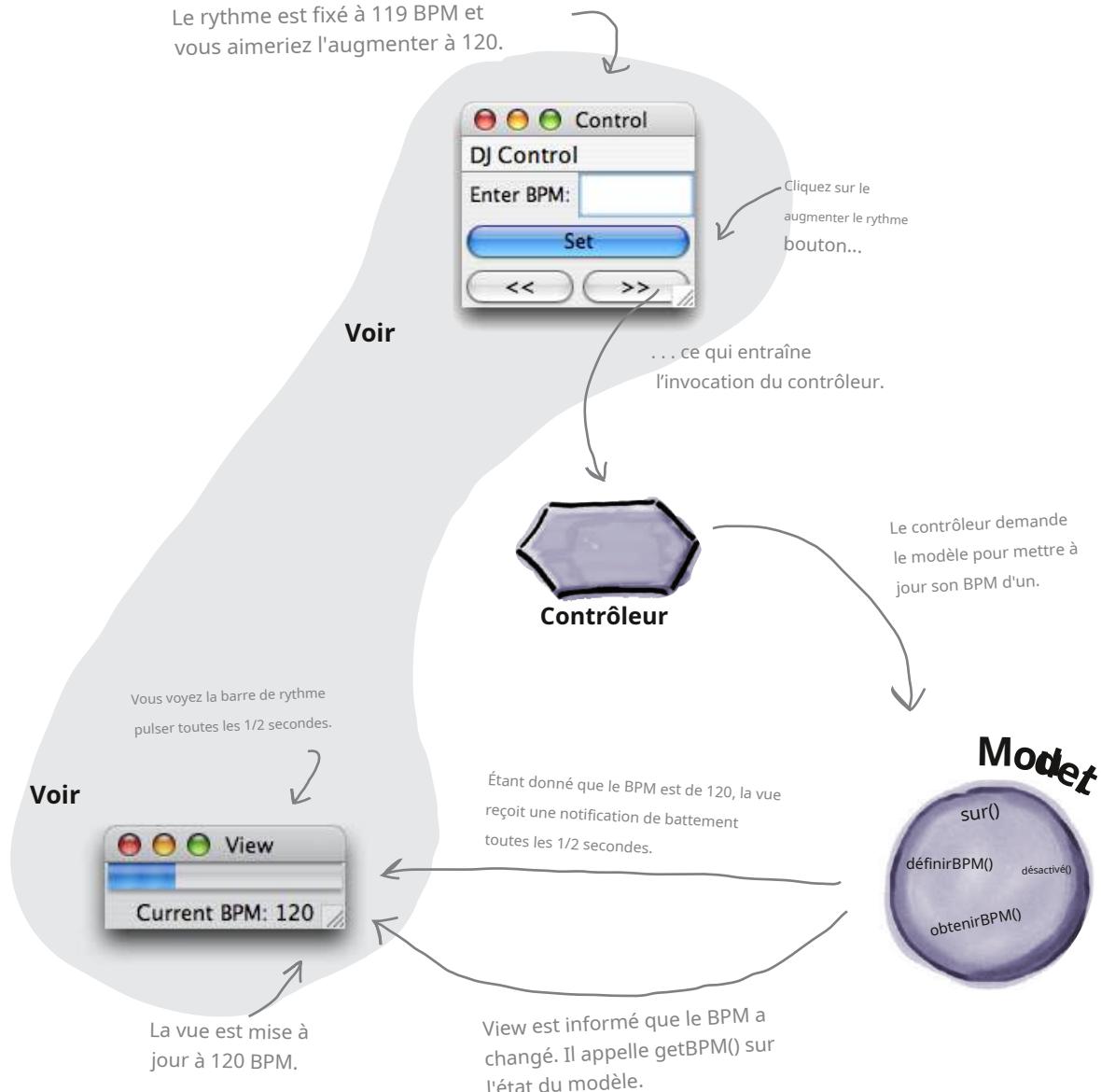
Le BeatModel est le cœur de l'application. Il implémente la logique permettant de démarrer et d'arrêter le rythme, de définir le BPM et de générer le son.

Le modèle nous permet également d'obtenir son état actuel grâce à la méthode getBPM().

Modèle



Rassembler les pièces ensemble



Construire les pièces

D'accord, vous savez que le modèle est responsable de la maintenance de toutes les données, de l'état et de toute logique d'application. Alors, qu'est-ce que le BeatModel contient ? Sa tâche principale est de gérer le rythme, il a donc un état qui maintient les battements actuels par minute et un code pour lire un clip audio afin de créer le rythme que nous entendons. Il expose également une interface qui permet au contrôleur de manipuler le rythme et permet à la vue et au contrôleur d'obtenir l'état du modèle. N'oubliez pas non plus que le modèle utilise le modèle Observer, nous avons donc également besoin de certaines méthodes pour permettre aux objets de s'enregistrer en tant qu'observateurs et d'envoyer des notifications.

Voyons l'interface BeatModelInterface avant de regarder l'implémentation :



Regardons maintenant la classe concrète BeatModel

```
la classe publique BeatModel implémente BeatModelInterface, Runnable {
    Liste<BeatObserver> beatObservers = new ArrayList<BeatObserver>();
    Liste<BPMObserver> bpmObservers = new ArrayList<BPMObserver>(); int bpm =
90;
    Fil fil;
    boolean stop = false; Clip
    clip;
```

Nous les utilisons pour démarrer et arrêter le fil de battement.

Voici le clip audio que nous jouons pour le beat.

Nous mettons en œuvre le BeatModelInterface et Runnable.

```
public void initialiser() {
```

```
    essayer {
        Ressource de fichier = nouveau fichier("clap.wav");
        clip = (Clip) AudioSystem.getLine(new Line.Info(Clip.class));
        clip.open(AudioSystem.getAudioInputStream(resource));
    }
    catch(Exception ex) {/* ... */}
}
```

Ces listes contiennent les deux types d'observateurs (observateurs Beat et BPM).

La variable bpm contient la fréquence des battements — par défaut, 90 BPM.

Cette méthode permet de configurer la piste rythmique.

```
public void sur() {
```

```
    bpm = 90;
    notifierBPMObservers();
    thread = nouveau Thread(ceci);
    stop = faux;
    thread.start();
}
```

La méthode on() définit les BPM par défaut et démarre le thread pour lire le rythme.

```
public void off() {
```

```
    arrêterBeat();
    arrêt = vrai;
}
```

Et off() l'arrête en définissant les BPM sur 0 et en arrêtant le thread de jouer le rythme.

```
public void run() {
```

```
    pendant que (!stop) {
        jouerBeat();
        notifierBeatObservers();
        essayer {
            Thread.sleep(60000/getBPM());
        }
        catch (Exception e) {}
    }
}
```

La méthode run() exécute le thread beat, joue un beat déterminé par le BPM et notifie les observateurs de beat qu'un beat a été joué. La boucle se termine lorsque nous sélectionnons Stop dans le menu.

```
public void setBPM(int bpm) {
    ceci.bpm = bpm;
    notifierBPMObservers();
}
```

La méthode setBPM() est la façon dont le contrôleur manipule le rythme. Elle définit la variable bpm et informe tous les observateurs de BPM que le BPM a changé.

```
public int getBPM() {
    retourner bpm;
}
```

La méthode getBPM() renvoie simplement les battements actuels par minute.



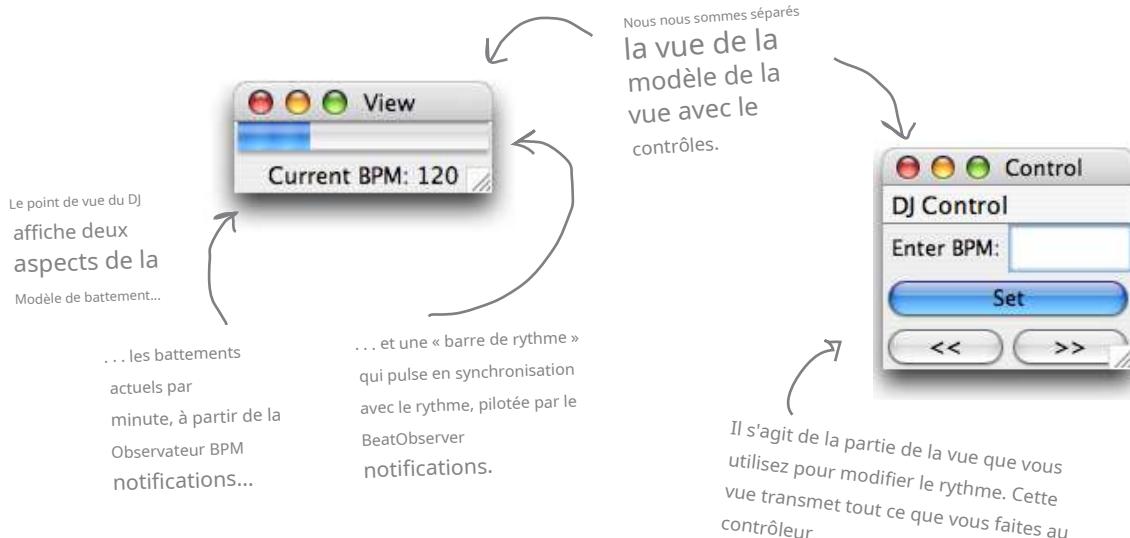
Code de cuisson prêt à l'emploi
Ce modèle utilise un clip audio pour générer des rythmes. Vous pouvez consulter l'intégralité mise en œuvre de tous les classes DJ dans les fichiers sources Java, disponibles sur wickedlysmart.com site, ou regardez le code à la fin du chapitre.

```
}
```

La vue

Maintenant, le plaisir commence ; nous pouvons connecter une vue et visualiser le BeatModel !

La première chose à remarquer à propos de la vue est que nous l'avons implémentée de manière à ce qu'elle soit affichée dans deux fenêtres distinctes. Une fenêtre contient le BPM actuel et le pouls ; l'autre contient les commandes de l'interface. Pourquoi ? Nous voulions souligner la différence entre l'interface qui contient la vue du modèle et le reste de l'interface qui contient l'ensemble des commandes utilisateur. Examinons de plus près les deux parties de la vue :



Notre BeatModel ne fait aucune hypothèse sur la vue. Le modèle est implanté à l'aide du modèle Observer, il informe donc simplement toute vue enregistrée en tant qu'observateur lorsque son état change. La vue utilise l'API du modèle pour accéder à l'état. Nous avons implanté un type de vue ; pouvez-vous penser à d'autres vues qui pourraient utiliser les notifications et l'état dans le BeatModel ?

Un spectacle de lumière basé sur le rythme en temps réel.

Une vue textuelle qui affiche un genre musical en fonction du BPM (ambient, downbeat, techno, etc.).

Mise en œuvre de la vue

Les deux parties de la vue (la vue du modèle et la vue avec les commandes de l'interface utilisateur) sont affichées dans deux fenêtres, mais cohabitent dans une même classe Java. Nous allons d'abord vous montrer uniquement le code qui crée la vue du modèle, qui affiche le BPM actuel et la mesure de temps. Ensuite, nous reviendrons sur la page suivante et vous montrerons uniquement le code qui crée les commandes de l'interface utilisateur, qui affichent le champ de saisie de texte du BPM et les boutons.



**Le code sur ces deux pages
n'est qu'un aperçu !**

Watch it! Ce que nous avons fait ici est de diviser UNE classe en DEUX, vous montrant une partie de la vue sur cette page et l'autre partie sur la page suivante. Tout ce code se trouve en réalité dans UNE classe : `DJView.java`. Tout est répertorié à la fin du chapitre.

DJView est un observateur des rythmes en temps réel et des changements de BPM.

la classe publique DJView implémente ActionListener,

```
Modèle BeatModelInterface ;  
Contrôleur ControllerInterface ;  
JFrame viewFrame ;  
Vue JPanelPanel;  
  
Barre de battements Barre de battements;  
JLabel bpmOutputLabel;
```

Ici, nous créons quelques composants pour l'affichage

Observateur de rythme, Observateur de BPM {

La vue contient une référence au modèle et au contrôleur. Le contrôleur n'est utilisé que par l'interface de contrôle, que nous aborderons dans une seconde...

```
public DJView(contrôleur ControllerInterface, modèle BeatModelInterface) {  
    this.controller = contrôleur;  
    this.model = modèle;  
    modèle.registerObserver((BeatObserver)this);  
    modèle.registerObserver((BPMObserver)this);  
}
```



Le cons
au cont
stocko
les vari

```
public void createView() {  
    // Créez tous les composants Swing ici  
}
```

Le constructeur obtient une référence au contrôleur et au modèle, et nous stockons les références à ceux-ci dans les variables d'instance.

Nous nous enregistrons également comme BeatObserver et BPMObserver du module

```
public void updateBPM() {  
    int bpm = modèle.getBPM(); si  
    (bpm == 0) {  
        bpmOutputLabel.setText("hors ligne"); }  
    else {  
        bpmOutputLabel.setText("BPM actuel : " + model.getBPM());  
    }  
}
```

La méthode `updateBPM()` est appelée lorsqu'un changement d'état se produit dans le modèle. Lorsque cela se produit, nous mettons à jour l'affichage avec le BPM actuel. Nous pouvons obtenir cette valeur en la demandant directement au modèle.

```
    public void updateBar()
    {
        beatBar();
    }
}
```

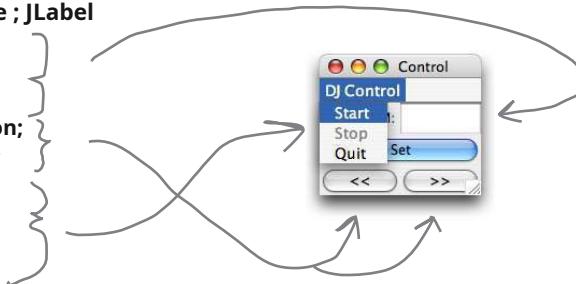
De même, la méthode updateBeat() est appelée lorsque le modèle démarre un nouveau battement. Lorsque cela se produit, nous devons pulser notre barre de battement. Pour ce faire, nous la définissons sur sa valeur maximale (100) et la laissons gérer l'animation de l'impulsion.

Mise en œuvre de la vue, suite...

Nous allons maintenant examiner le code de la partie contrôles de l'interface utilisateur de la vue. Cette vue vous permet de contrôler le modèle en indiquant au contrôleur ce qu'il doit faire, ce qui indique à son tour au modèle ce qu'il doit faire. N'oubliez pas que ce code se trouve dans le même fichier de classe que l'autre code de la vue.

la classe publique DJView implémente ActionListener, BeatObserver, BPMObserver {

```
Modèle BeatModelInterface ;
Contrôleur ControllerInterface ; JLabel
bpmLabel ;
JTextField bpmTextField;
JButton setBPMButton;
JButton augmenter le BPMButton;
JButton diminuer le BPMButton;
JMenuBar menuBar;
Menu JMenu;
 JMenuItem startMenuItem;
 JMenuItem stopMenuItem;
```



```
public void createControls() {
    // Créez tous les composants Swing ici
}
```

Cette méthode crée tous les contrôles et les place dans l'interface. Elle s'occupe également du menu. Lorsque les éléments d'arrêt ou de démarrage sont choisis, les méthodes correspondantes sont appelées sur le contrôleur.

```
public void enableStopMenuItem() {
    stopMenuItem.setEnabled(true);
}
```

Toutes ces méthodes permettent d'activer et de désactiver les éléments de démarrage et d'arrêt du menu. Nous verrons que le contrôleur les utilise pour modifier l'interface.

```
public void désactiverStopMenuItem() {
    stopMenuItem.setEnabled(false);
}
```

```
public void enableStartMenuItem() {
    startMenuItem.setEnabled(true);
}
```

Cette méthode est appelée lorsqu'un bouton est cliqué.

```
public void désactiverStartMenuItem() {
    startMenuItem.setEnabled(false);
}
```

Si le bouton Set est cliqué, il est transmis au contrôleur avec le nouveau bpm.

```
public void actionPerformed(événement ActionEvent) {
    si (event.getSource() == setBPMButton) {
        int bpm = Integer.parseInt(bpmTextField.getText());
        contrôleur.setBPM(bpm);
    } sinon si (event.getSource() == increaseBPMButton) {
        contrôleur.augmenteBPM();
    } sinon si (event.getSource() == diminuerBPMButton) {
        contrôleur.diminueBPM();
    }
}
```

De même, si le bouton d'augmentation ou de diminution est cliqué, cette information est transmise au contrôleur.

Maintenant pour le contrôleur

Il est temps d'écrire la pièce manquante : le contrôleur. Rappelez-vous que le contrôleur est la stratégie que nous insérons dans la vue pour lui donner un peu d'intelligence.

Comme nous implémentons le modèle de stratégie, nous devons commencer par une interface pour toute stratégie susceptible d'être connectée à la vue DJ. Nous allons l'appeler ControllerInterface.

```
interface publique ControllerInterface {  
    void démarrer();  
    void arrêter();  
    void augmenterBPM();  
    void diminuerBPM();  
    void setBPM(int bpm);  
}
```

Voici toutes les méthodes que la vue peut appeler sur le contrôleur.



Ces éléments devraient vous sembler familiers après avoir vu l'interface du modèle. Vous pouvez arrêter et démarrer la génération de rythmes et modifier le BPM. Cette interface est « plus riche » que l'interface BeatModel car vous pouvez ajuster les BPM en les augmentant et en les diminuant.



Puzzle de conception

Vous avez vu que la vue et le contrôleur utilisent ensemble le modèle de stratégie. Pouvez-vous dessiner un diagramme de classes des deux qui représente ce modèle ?

Et voici l'implémentation du contrôleur :

la classe publique BeatController implémente ControllerInterface {

Modèle BeatModelInterface ;

vue DJView ;

public BeatController(BeatModelInterface modèle) {

ce.modèle = modèle;

vue = nouveau DJView(ceci, modèle);

vue.createView();

vue.createControls();

vue.disableStopMenuItem();

vue.enableStartMenuItem();

modèle.initialize();

}

public void start() {

modèle.on();

afficher.disableStartMenuItem();

afficher.enableStopMenuItem();

}

public void stop() {

modèle.off();

afficher.disableStopMenuItem();

afficher.enableStartMenuItem();

}

public void increaseBPM() {

int bpm = modèle.getBPM();

modèle.setBPM(bpm + 1);

}

public void diminutionBPM() {

int bpm = modèle.getBPM();

modèle.setBPM(bpm - 1);

}

public void setBPM(int bpm) {

modèle.setBPM(bpm);

}

Le contrôleur implémente le ControllerInterface.

Le contrôleur est la substance crémeuse au milieu du cookie MVC Oreo, c'est donc l'objet qui maintient la vue et le modèle et colle le tout ensemble.

Le contrôleur reçoit le modèle dans le constructeur et crée ensuite la vue.

Lorsque vous choisissez Démarrer dans le menu de l'interface utilisateur, le contrôleur allume le modèle, puis modifie l'interface utilisateur de sorte que l'élément de menu Démarrer soit désactivé et que l'élément de menu Arrêter soit activé.

De même, lorsque vous choisissez Arrêter dans le menu, le contrôleur éteint le modèle et modifie l'interface utilisateur de sorte que l'élément de menu Arrêter soit désactivé et l'élément de menu Démarrer soit activé.

Si le bouton d'augmentation est cliqué, le contrôleur obtient le BPM actuel du modèle, en ajoute un, puis définit un nouveau BPM.

Même chose ici, sauf qu'on soustrait un au BPM actuel.

Enfin, si l'interface utilisateur est utilisée pour définir un BPM arbitraire, le contrôleur demande au modèle de définir son BPM.

REMARQUE : le contrôleur effectue l'intelligent décisions pour la vue. La vue sait juste comment activer et désactiver les éléments de menu ; elle ne connaît pas les situations dans lesquelles elle doit les désactiver.

Mettre tout cela ensemble...

Nous avons tout ce dont nous avons besoin : un modèle, une vue et un contrôleur. Il est maintenant temps de les assembler ! Nous allons voir et entendre à quel point ils fonctionnent bien ensemble.

Tout ce dont nous avons besoin est un peu de code pour démarrer ; cela ne prendra pas beaucoup de temps :



```
classe publique DJTestDrive {
```

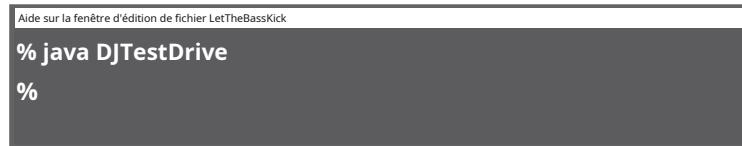
```
    public static void main (String[] args) {
        BeatModelInterface modèle = nouveau BeatModel(); ControllerInterface
        contrôleur = nouveau BeatController(modèle);
    }
}
```

Créez d'abord un modèle...

... puis créez un contrôleur et
transmettez-lui le modèle. N'oubliez pas
que le contrôleur crée la vue, nous
n'avons donc pas besoin de le faire.

Et maintenant, un essai...

Assurez-vous d'avoir le
fichier clip.wav au
niveau supérieur du
dossier de code !



Exécutez ceci...

... et vous verrez cela.



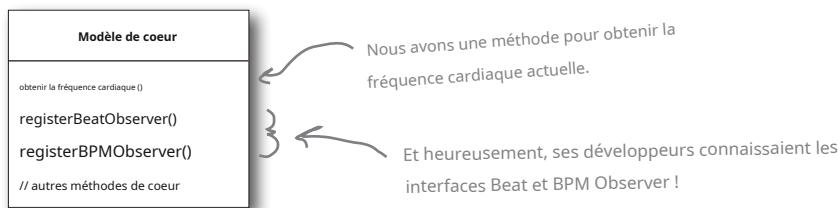
Choses à essayer

- ➊ Démarrez la génération de rythme avec l'élément de menu Démarrer ; notez que le contrôleur désactive l'élément par la suite.
- ➋ Utilisez la saisie de texte ainsi que les boutons d'augmentation et de diminution pour modifier le BPM. Remarquez comment l'affichage reflète les modifications malgré le fait qu'il n'a aucun lien logique avec les commandes.
- ➌ Remarquez comment la barre de rythme suit toujours le rythme puisqu'elle est un observateur du modèle.
- ➍ Mettez votre chanson préférée et voyez si vous pouvez suivre le rythme en utilisant les commandes d'augmentation et de diminution.
- ➎ Arrêtez le générateur. Notez comment le contrôleur désactive l'élément de menu Arrêter et active l'élément de menu Démarrer.

Exploration de la stratégie

Examinons un peu plus en détail le modèle de stratégie pour mieux comprendre comment il est utilisé dans MVC. Nous allons également voir apparaître un autre modèle convivial, un modèle que vous verrez souvent traîner dans le trio MVC : le modèle d'adaptateur.

Pensez une seconde à ce que fait la vue DJ : elle affiche un rythme cardiaque et un pouls. Cela vous rappelle-t-il autre chose ? Et pourquoi pas un battement de cœur ? Il se trouve que nous avons une classe de moniteur cardiaque ; voici le diagramme de classe :



Il serait certainement intéressant de réutiliser notre vue actuelle avec le HeartModel, mais nous avons besoin d'un contrôleur qui fonctionne avec ce modèle. De plus, l'interface du HeartModel ne correspond pas à ce que la vue attend car elle possède une méthode getHeartRate() plutôt qu'une méthode getBPM(). Comment concevriez-vous un ensemble de classes pour permettre à la vue d'être réutilisée avec le nouveau modèle ? Notez vos idées de conception de classe ci-dessous.

mvc et adaptateur

Adaptation du modèle

Pour commencer, nous allons devoir adapter le HeartModel à un BeatModel. Si nous ne le faisons pas, la vue ne pourra pas fonctionner avec le modèle, car la vue ne sait que comment obtenir BPM(), et la méthode équivalente du modèle cardiaque est getHeartRate(). Comment allons-nous procéder ? Nous allons bien sûr utiliser le modèle Adapter ! Il s'avère qu'il s'agit d'une technique courante lorsque l'on travaille avec MVC : utiliser un adaptateur pour adapter un modèle afin qu'il fonctionne avec des contrôleur et des vues existants.

Voici le code pour adapter un HeartModel en BeatModel :

```
la classe publique HeartAdapter implémente BeatModelInterface {
    HeartModelInterface coeur;
```

Nous devons implémenter l'interface cible — dans ce cas, BeatModelInterface.

```
public HeartAdapter(HeartModelInterface coeur) {
    ceci.coeur = coeur;
}
```

Ici, nous stockons une référence au modèle de cœur.

```
public void initialize() {}
```

Nous ne savons pas ce que ces opérations pourraient faire à un cœur, mais cela semble effrayant. Nous les laisserons donc comme « sans opération ».

```
public void on() {}
```

```
public void off() {}
```

```
public int getBPM() {
    retourner coeur.getHeartRate();
}
```

Lorsque getBPM() est appelé, nous le traduirons simplement en un appel getHeartRate() sur le modèle cardiaque.

```
public void setBPM(int bpm) {}
```

Nous ne voulons pas faire ça sur un cœur ! Encore une fois, laissons cela comme « pas d'opération ».

```
public void registerObserver(BeatObserver o) {
    coeur.registerObserver(o);
}
```

Voici nos méthodes d'observation. Nous les déléguons simplement au modèle du cœur enveloppé.

```
public void removeObserver(BeatObserver o) {
    coeur.removeObserver(o);
}
```

```
public void registerObserver(BPMObserver o) {
    coeur.registerObserver(o);
}
```

```
}
```

Nous sommes maintenant prêts pour un HeartController

Avec notre HeartAdapter en main, nous devrions être prêts à créer un contrôleur et à exécuter la vue avec le HeartModel. Parlons de réutilisation !

```
la classe publique HeartController implémente ControllerInterface {
    Modèle HeartModelInterface ;
    Vue DJView ;

    public HeartController (modèle HeartModelInterface) {
        ce.modèle = modèle;
        vue = nouveau DJView(ceci, nouveau HeartAdapter(modèle));
        vue.createView();
        vue.createControls();
        afficher.disableStopMenuItem();
        afficher.disableStartMenuItem();
    }

    public void start() {}

    public void stop() {}

    public void increaseBPM() {}

    public void diminutionBPM() {}

    public void setBPM(int bpm) {}
}
```

Le HeartController implémente le ControllerInterface, tout comme le BeatController l'a fait.

Comme avant, le contrôleur crée la vue et colle tout ensemble.

Il y a un changement : on nous passe un HeartModel, pas un BeatModel...

... et nous devons envelopper ce modèle avec un adaptateur avant de le transmettre à la vue.

Enfin, le HeartController désactive les éléments de menu car ils ne sont pas nécessaires.

Il n'y a pas grand-chose à faire ici ; après tout, nous ne pouvons pas vraiment contrôler les coeurs comme nous pouvons battre les machines.

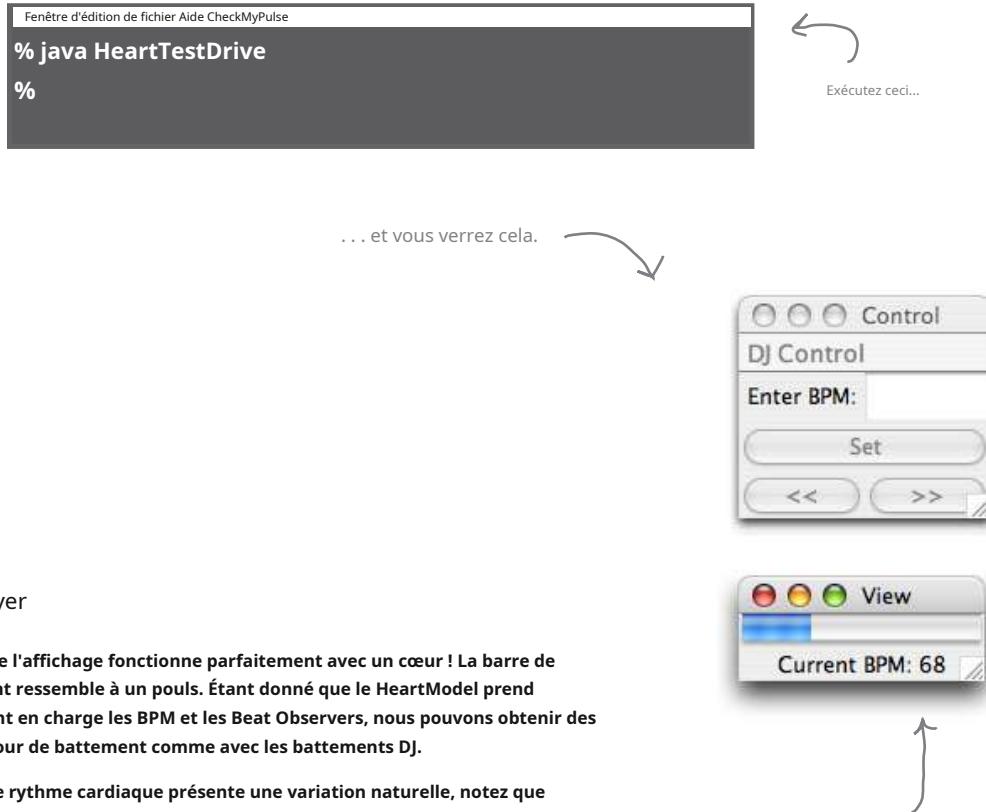
Et voilà ! Il est maintenant temps de passer à un peu de code de test...

```
classe publique HeartTestDrive {
```

```
public static void main (String[] args) {
    HeartModel heartModel = new HeartModel(); Modèle ControllerInterface =
    new HeartController (heartModel);
}
```

Tout ce que nous devons faire est de créer le contrôleur et de lui transmettre un moniteur cardiaque.

Et maintenant, un essai...



Choses à essayer

- 1 Notez que l'affichage fonctionne parfaitement avec un cœur ! La barre de battement ressemble à un pouls. Étant donné que le HeartModel prend également en charge les BPM et les Beat Observers, nous pouvons obtenir des mises à jour de battement comme avec les battements DJ.
- 2 Comme le rythme cardiaque présente une variation naturelle, notez que l'affichage est mis à jour avec les nouveaux battements par minute.
- 3 Chaque fois que nous recevons une mise à jour BPM, l'adaptateur fait son travail de traduction des appels getBPM() en appels getHeartRate().
- 4 Les éléments de menu Démarrer et Arrêter ne sont pas activés car le contrôleur les a désactivés.
- 5 Les autres boutons fonctionnent toujours mais n'ont aucun effet car le contrôleur n'implémente aucune opération pour eux. La vue pourrait être modifiée pour prendre en charge la désactivation de ces éléments.

there are no Dumb Questions

Q: Il semble que vous fassiez vraiment semblant du fait que le modèle composite est réellement présent dans MVC. Est-il vraiment présent ?

UN: Oui, Virginia, il existe vraiment un modèle composite dans MVC. Mais, en fait, c'est une très bonne question. Aujourd'hui, les packages d'interface utilisateur graphique, comme Swing, sont devenus si sophistiqués que nous remarquons à peine la structure interne et l'utilisation de Composite dans la construction et la mise à jour de l'affichage. C'est encore plus difficile à voir lorsque nous avons des navigateurs Web qui peuvent prendre un langage de balisage et le convertir en une interface utilisateur.

À l'époque où MVC a été découvert, la création d'interfaces graphiques nécessitait beaucoup plus d'interventions manuelles et le modèle faisait plus évidemment partie du MVC.

Q: Le contrôleur implémente-t-il une logique d'application ?

UN: Non, le contrôleur implémente le comportement de la vue. C'est l'intelligence qui traduit les actions de la vue en actions sur le modèle. Le modèle prend ces actions et implémente la logique d'application pour décider quoi faire en réponse à ces actions. Le contrôleur devra peut-être effectuer un peu de travail pour déterminer les appels de méthode à effectuer sur le modèle, mais cela n'est pas considéré comme la « logique d'application ». La logique d'application est le code qui gère et manipule vos données et qui réside dans votre modèle.

Q: J'ai toujours eu du mal à comprendre le mot « modèle ». Je comprends maintenant qu'il s'agit de l'essentiel de l'application, mais pourquoi un mot aussi vague et difficile à comprendre a-t-il été utilisé pour décrire cet aspect de MVC ?

UN: Lorsqu'ils ont nommé MVC, ils avaient besoin d'un mot commençant par un « M », sinon ils n'auraient pas pu l'appeler MVC.

Mais sérieusement, nous sommes d'accord avec vous. Tout le monde se gratte la tête et se demande ce qu'est un modèle. Mais ensuite, tout le monde se rend compte qu'il n'arrive pas non plus à trouver un meilleur mot.

Q: Vous avez beaucoup parlé de l'état du modèle. Cela signifie-t-il qu'il contient le modèle d'état ?

UN: Non, nous parlons de l'idée générale d'état. Mais certains modèles utilisent certainement le modèle d'état pour gérer leurs états internes.

Q: J'ai vu des descriptions de MVC où le contrôleur est décrit comme un « médiateur » entre la vue et le modèle. Le contrôleur implémente-t-il le modèle de médiateur ?

UN: Nous n'avons pas abordé le modèle de médiateur (bien que vous trouviez un résumé du modèle dans l'annexe), nous n'entrerons donc pas trop dans les détails ici, mais l'objectif du médiateur est d'encapsuler la manière dont les objets interagissent et de promouvoir un couplage lâche en empêchant deux objets de se référer explicitement l'un à l'autre. Ainsi, dans une certaine mesure, le contrôleur peut être considéré comme un médiateur, puisque la vue ne définit jamais l'état directement sur le modèle, mais passe toujours par le contrôleur. N'oubliez pas, cependant, que la vue a une référence au modèle pour accéder à son état. Si le contrôleur était vraiment un médiateur, la vue devrait également passer par le contrôleur pour obtenir l'état du modèle.

Q: La vue doit-elle toujours demander au modèle son état ? Ne pourrait-on pas utiliser le modèle push et envoyer l'état du modèle avec la notification de mise à jour ?

UN: Oui, le modèle pourrait certainement envoyer son état avec la notification, et nous pourrions faire quelque chose de similaire avec le BeatModel en envoyant uniquement l'état qui intéresse la vue. Si vous vous souvenez du chapitre sur le modèle Observer, vous vous souviendrez également qu'il y a quelques inconvénients à cela. Si ce n'est pas le cas, revenez au chapitre 2 et regardez-le de plus près. Le modèle MVC a été adapté à un certain nombre de modèles similaires, en particulier pour l'environnement navigateur/serveur du Web; vous trouverez donc de nombreuses exceptions à la règle.

Q: Si j'ai plusieurs vues, ai-je toujours besoin de plusieurs contrôleurs ?

UN: En règle générale, vous avez besoin d'un contrôleur par vue lors de l'exécution ; cependant, la même classe de contrôleur peut facilement gérer plusieurs vues.

Q: La vue n'est pas censée manipuler le modèle. Cependant, j'ai remarqué dans votre implémentation que la vue a un accès complet aux méthodes qui modifient l'état du modèle. Est-ce dangereux ?

UN: Vous avez raison ; nous avons donné à la vue un accès complet à l'ensemble des méthodes du modèle. Nous l'avons fait pour simplifier les choses, mais il peut y avoir des circonstances dans lesquelles vous souhaitez donner à la vue un accès à une partie seulement de l'API de votre modèle. Il existe un excellent modèle de conception qui vous permet d'adapter une interface pour ne fournir qu'un sous-ensemble. Pouvez-vous y penser ?



La plupart de mes utilisateurs
les interfaces sont
en fait, basé sur un
navigateur. Est-ce que tout
cela va m'aider ?

Oui!

Le modèle MVC est si utile qu'il a été adapté à de nombreux frameworks Web. Bien entendu, le Web fonctionne différemment de votre application standard, il existe donc plusieurs approches différentes pour appliquer le modèle MVC au Web.

Les applications Web ont un côté client (le navigateur) et un côté serveur. Cela étant dit, nous pouvons faire différents compromis de conception en fonction de l'emplacement du modèle, de la vue et du contrôleur. *client léger* approches, le modèle, la plupart de la vue et le contrôleur résident tous dans le serveur, le navigateur fournissant un moyen d'afficher la vue et d'obtenir des entrées du navigateur vers le contrôleur. Une autre approche est la *application d'une seule page*, où la quasi-totalité du modèle, de la vue et du contrôleur résident du côté client. Ce sont les deux extrémités du spectre, et vous trouverez des frameworks qui varient la mesure dans laquelle chaque composant (c'est-à-dire le modèle, la vue et le contrôleur) réside sur le client ou le serveur, ainsi que des modèles hybrides où certains composants sont partagés entre le client et le serveur.

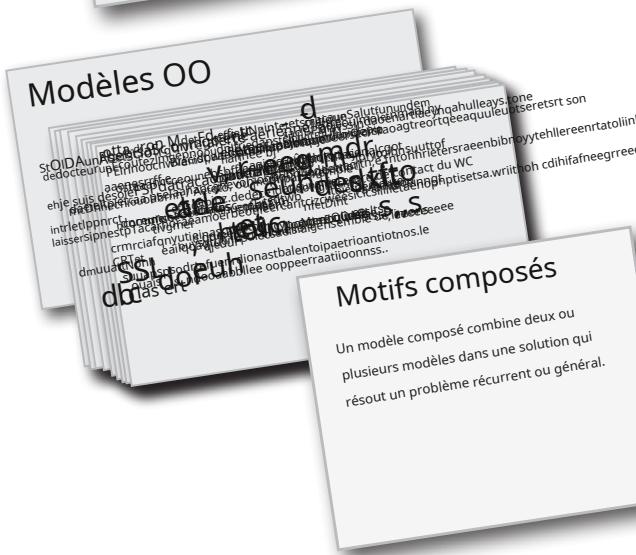
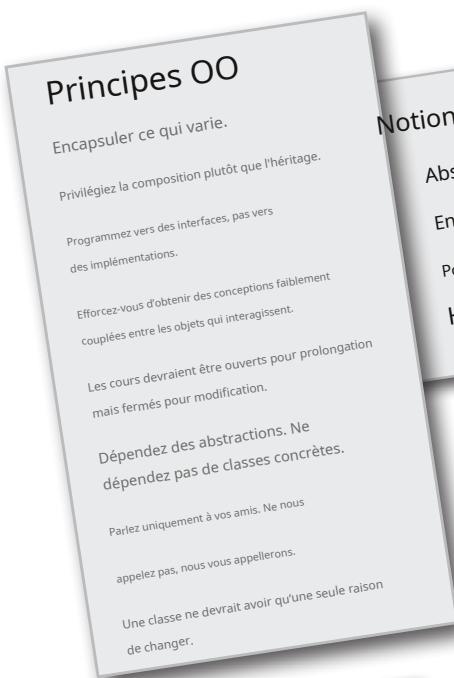
Il existe de nombreux frameworks Web MVC populaires, comme Spring Web MVC, Django, ASP.NET MVC, AngularJS, EmberJS, JavaScriptMVC, Backbone, et sans doute d'autres encore à venir. Dans la plupart des cas, chaque framework a sa propre façon unique de mapper le modèle, la vue et le contrôleur sur le client et le serveur. Maintenant que vous connaissez le modèle MVC, vous n'aurez aucun problème à adapter vos connaissances au framework que vous choisissez d'utiliser.



Des outils pour votre boîte à outils de conception

Vous pourriez impressionner n'importe qui avec votre boîte à outils de conception.

Waouh, regardez tous ces principes, ces modèles et maintenant, ces modèles composés !



BULLET POINTS

- f Le modèle MVC (Model View Controller) est un modèle composé composé de l'observateur, de la stratégie et Motifs composites.
- f Le modèle utilise le modèle Observer afin de pouvoir tenir les observateurs à jour tout en restant découpé d'eux.
- f Le contrôleur est la stratégie de la vue. La vue peut utiliser différentes implémentations du contrôleur pour obtenir un comportement différent.
- f La vue utilise le modèle composite pour implémenter l'interface utilisateur, qui se compose généralement de composants imbriqués tels que des panneaux, des cadres et des boutons.
- f Ces modèles fonctionnent ensemble pour découpler les trois acteurs du modèle MVC, ce qui permet de conserver des conceptions claires et flexibles.
- f Le modèle d'adaptateur peut être utilisé pour adapter un nouveau modèle à une vue et un contrôleur existants.
- f MVC a été adapté au Web.
- f Il existe de nombreux frameworks Web MVC avec diverses adaptations du modèle MVC pour s'adapter à la structure de l'application client/serveur.



Solutions d'exercices



Le QuackCounter est également un Quackable. Lorsque nous modifions Quackable pour étendre QuackObservable, nous devons modifier chaque classe qui implémente Quackable, y compris QuackCounter :

QuackCounter est un Quackable, donc maintenant c'est aussi un QuackObservable.

la classe publique QuackCounter implémente Quackable {

 Canard cananant;

 nombre statique intOfQuacks;

 public QuackCounter(canard canetonnant) {

 ce.canard = canard;

 }

 public void charlatan() {

 canard.coin-coin();

 nombreDeCharlatanismes++;

 }

 public static int getQuacks() {

 retourner nombreDeQuacks;

 }

Voici le canard que QuackCounter est en train de décorer. C'est ce canard qui doit réellement gérer les méthodes observables.

Tout ce code est le même que la version précédente de QuackCounter.

 public void registerObserver(Observateur observateur) {

 canard.registerObserver(observateur);

 }

 public void notifierObservers() {

 canard.notifyObservers();

 }

Voici les deux QuackObservable méthodes. Notez que nous déléguons simplement les deux appels au canard que nous décorons.



Sharpen your pencil Solution

Et si notre charlatan veut observer un troupeau entier ? Qu'est-ce que cela signifie ? Pensez-y comme ceci : si nous observons un composite, alors nous observons tout ce qui se trouve dans le composite. Ainsi, lorsque vous vous enregistrez dans un troupeau, le composite du troupeau s'assure que vous êtes enregistré avec tous ses enfants, ce qui peut inclure d'autres troupeaux.

la classe publique Flock implémente Quackable {

```
Liste<Quackable> quackers = new ArrayList<Quackable>();
```

```
public void add(canard canasson) {
    canards.add(canard);
}
```

Flock est un Quackable, donc maintenant c'est aussi un QuackObservable.

```
public void charlatan() {
    Itérateur<Quackable> itérateur = quackers.iterator(); while
    (itérateur.hasNext()) {
        Canard cancanant = itérateur.next();
        duck.quack();
    }
}
```

Voici les charlatans qui sont dans le troupeau.

```
public void registerObserver(Observateur observateur) {
    Itérateur<Quackable> itérateur = canards.iterator(); while
    (itérateur.hasNext()) {
        Canard cancanant = itérateur.next();
        duck.registerObserver(observer);
    }
}
```

Lorsque vous vous inscrivez en tant qu'observateur auprès du troupeau, vous êtes en fait enregistré avec tout ce qui se trouve dans le troupeau, c'est-à-dire tous les cancaniers, qu'il s'agisse d'un canard ou d'un autre troupeau.

```
public void notifierObservers() {}
```

Nous parcourons tous les Quackables du Flock et déléguons l'appel à chaque Quackable. Si le Quackable est un autre Flock, il fera de même.

```
}
```

Chaque Quackable fait sa propre notification, donc Flock n'a pas à s'en soucier. Cela se produit lorsque Flock délègue quack() à chaque Quackable du Flock.



Sharpen your pencil Solution

Nous instancions toujours directement Geese en nous appuyant sur des classes concrètes.
Pouvez-vous écrire une fabrique abstraite pour Geese ? Comment devrait-elle gérer la création de « canards oies » ?

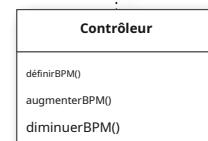
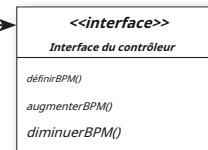
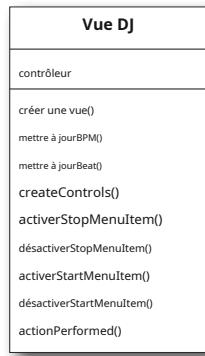
Vous pouvez ajouter une méthode `createGooseDuck()` aux usines de canards existantes. Vous pouvez également créer une usine complètement distincte pour créer des familles d'oies.



Solution du puzzle de conception

Vous avez vu que la vue et le contrôleur utilisent ensemble le modèle de stratégie. Pouvez-vous dessiner un diagramme de classes des deux qui représente ce modèle ?

Les délégués de vue comportement à l'égard de contrôleur. Le comportement il les délégués sont la façon de contrôler le modèle basé sur l'utilisateur saisir.



Le Interface du contrôleur est l'interface que tout est concret contrôleurs mettre en œuvre. Ceci est la stratégie interface.

Nous pouvons brancher dans différents contrôleurs fournir différent comportements pour la vue.



Prêt à cuire Code

Voici l'implémentation complète de DJView. Elle montre tout le code MIDI pour générer le son et tous les composants Swing pour créer la vue. Vous pouvez également télécharger ce code sur <https://www.wickedlysmart.com>. Amusez-vous bien !

```
paquet headfirst.designpatterns.combined.djview;
```

```
classe publique DJTestDrive {
```

```
    public static void main (String[] args) {
        BeatModelInterface modèle = nouveau BeatModel(); ControllerInterface
        contrôleur = nouveau BeatController(modèle);
    }
}
```

Le modèle Beat

```
paquet headfirst.designpatterns.combined.djview;
```

```
interface publique BeatModelInterface {
    void initialiser();

    vide sur();

    annuler désactivé();

    void setBPM(int bpm);

    int getBPM();

    void registerObserver(BeatObserver o);

    void removeObserver(BeatObserver o);

    void registerObserver(BPMObserver o);

    void removeObserver(BPMObserver o);
}
```

```
paquet headfirst.designpatterns.combined.djview;

importer java.util.*;
importer javax.sound.sampled.AudioSystem;
importer javax.sound.sampled.Clip; importer
java.io.*;
importer javax.sound.sampled.Line;

la classe publique BeatModel implémente BeatModelInterface, Runnable {
    Liste<BeatObserver> beatObservers = new ArrayList<BeatObserver>();
    Liste<BPMObserver> bpmObservers = new ArrayList<BPMObserver>(); int bpm =
90;
    Fil fil;
    booléen stop = false; Clip
clip;

    public void initialiser() {
        essayer {
            Ressource de fichier = nouveau fichier("clap.wav");
            clip = (Clip) AudioSystem.getLine(new Line.Info(Clip.class));
            clip.open(AudioSystem.getAudioInputStream(resource));
        }
        attraper(Exception ex) {
            System.out.println("Erreur : Impossible de charger le
clip"); System.out.println(ex);
        }
    }

    public void sur() {
        bpm = 90;
        notifierBPMObservers();
        thread = nouveau Thread(ceci);
        stop = faux;
        thread.start();
    }

    public void off() {
        arrêterBeat();
        arrêt = vrai;
    }
}
```



Prêt à cuire Code

```

public void run() {
    pendant que (!stop) {
        jouerBeat();
        notifierBeatObservers();
        essayer {
            Thread.sleep(60000/getBPM());
        }
        catch (Exception e) {}
    }
}

public void setBPM(int bpm) {
    ceci.bpm = bpm;
    notifierBPMObservers();
}

public int getBPM() {
    retourner bpm;
}

public void registerObserver(BeatObserver o) {
    beatObservers.add(o);
}

public void notifierBeatObservers() {
    pour (int i = 0; i < beatObservers.size(); i++) {
        observer.updateBeat();
    }
}

public void registerObserver(BPMObserver o) {
    bpmObservers.add(o);
}

public void notifierBPMObservers() {
    pour (int i = 0; i < bpmObservers.size(); i++) {
        observer.updateBPM();
    }
}

```

```
public void removeObserver(BeatObserver o) {
    int i = beatObservers.indexOf(o); si (i >= 0)
    {
        beatObservers.remove(i);
    }
}

public void removeObserver(BPMObserver o) {
    int i = bpmObservers.indexOf(o); si (i >=
0) {
        bpmObservers.remove(i);
    }
}

public void playBeat() {
    clip.setFramePosition(0);
    clip.start();
}
public void stopBeat() {
    clip.setFramePosition(0);
    clip.stop();
}

}
```

La vue



```
paquet headfirst.designpatterns.combined.djview;
```

```
interface publique BeatObserver {
    void updateBeat();
}
```

```
paquet headfirst.designpatterns.combined.djview;
```

```
interface publique BPMObserver {
    void updateBPM();
}
```

```
paquet headfirst.designpatterns.combined.djview;
```

```
importer java.awt.*;
importer java.awt.event.*;
importer javax.swing.*;
```

```
la classe publique DJView implémente ActionListener, BeatObserver, BPMObserver {
```

```
    Modèle BeatModelInterface ;
    Contrôleur ControllerInterface ;
    JFrame viewFrame ;
    Vue JPanelPanel;
```

```
Barre de battements Barre de battements;
```

```
    JLabel bpmOutputLabel;
    Contrôle JFrameFrame;
    Panneau de contrôle JPanel ;
    JÉtiquette bpmÉtiquette;
    JTextField bpmTextField;
    JButton setBPMButton;
    JButton augmenter le BPMButton;
    JButton diminuer le BPMButton;
    JMenuBar menuBar;
    Menu JMenu;
    JMenuItem startMenuItem;
    JMenuItem stopMenuItem;
```

```
public DJView(contrôleur ControllerInterface, modèle BeatModelInterface) {
```

```
    this.controller = contrôleur;
    this.model = modèle;
    modèle.registerObserver((BeatObserver)this);
    modèle.registerObserver((BPMObserver)this);
```

```
}
```

```
public void createView() {
    // Créez tous les composants Swing ici
    JPanel(new GridLayout(1, 2)); viewFrame = new
    JFrame("View");
    viewFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    viewFrame.setSize(nouvelle dimension(100, 80));
    bpmOutputLabel = new JLabel("hors ligne", SwingConstants.CENTER); beatBar
    = nouveau BeatBar();
    beatBar.setValue(0);
    JPanel bpmPanel = new JPanel(new GridLayout(2, 1));
    bpmPanel.add(beatBar);
    bpmPanel.add(bpmOutputLabel); viewPanel.add(bpmPanel);
    viewFrame.getContentPane().add(viewPanel, BorderLayout.CENTER);
    viewFrame.pack();

    viewFrame.setVisible(true);
}

public void createControls() {
    // Créez tous les composants Swing ici
    JFrame.setDefaultLookAndFeelDecorated(true); controlFrame = new
    JFrame("Control");
    controlFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    controlFrame.setSize(new Dimension(100, 80));

    controlPanel = nouveau JPanel(nouveau GridLayout(1, 2));

    menuBar = new JMenuBar(); menu = new
    JMenu("Contrôle DJ"); startMenuItem = new
    JMenuItem("Démarrer");
    menu.add(startMenuItem);
    startMenuItem.addActionListener(new ActionListener() {
        public void actionPerformed(événement ActionEvent) {
            contrôleur.start();
        }
    });
    stopMenuItem = new JMenuItem("Arrêter");
    menu.add(stopMenuItem);
    stopMenuItem.addActionListener(nouveau ActionListener() {
        public void actionPerformed(événement ActionEvent) {
            contrôleur.stop();
        }
    });
    JMenuItem exit = new JMenuItem("Quitter");
    exit.addActionListener(new ActionListener() {
        public void actionPerformed(événement ActionEvent) {
            Système.exit(0);
        }
    });
}
```



Prêt à cuire Code

```

menu.add(sortie);
menuBar.add(menu);
controlFrame.setJMenuBar(menuBar);

bpmTextField = nouveau JTextField(2);
bpmLabel = new JLabel("Entrez le BPM :", SwingConstants.RIGHT);
setBPMBUTTON = new JButton("Définir");
setBPMBUTTON.setSize(nouvelle dimension(10,40));
increaseBPMBUTTON = nouveau JButton(">>");
increaseBPMBUTTON = nouveau JButton("<<");
setBPMBUTTON.addActionListener(this);
increaseBPMBUTTON.addActionListener(this);
increaseBPMBUTTON.addActionListener(this);

 JPanel buttonPanel = nouveau JPanel(nouveau GridLayout(1, 2));
buttonPanel.add(decreaseBPMBUTTON);
buttonPanel.add(augmenterBPMBUTTON);

 JPanel enterPanel = new JPanel(new GridLayout(1, 2));
enterPanel.add(bpmLabel);
enterPanel.add(bpmTextField);
 JPanel insideControlPanel = nouveau JPanel(nouveau GridLayout(3, 1));
insideControlPanel.add(enterPanel);
insideControlPanel.add(setBPMBUTTON);
insideControlPanel.add(buttonPanel);
controlPanel.add(insideControlPanel);

bpmOutputLabel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));

controlFrame.getRootPane().setDefaultButton(setBPMBUTTON);
controlFrame.getContentPane().add(controlPanel, BorderLayout.CENTER);

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

public void enableStopMenuItem() {
    stopMenuItem.setEnabled(true);
}

public void désactiverStopMenuItem() {
    stopMenuItem.setEnabled(faux);
}

```

code de cuisson prêt à l'emploi : contrôleur

```
public void enableStartMenuItem() {
    startMenuItem.setEnabled(true);
}

public void désactiverStartMenuItem() {
    startMenuItem.setEnabled(faux);
}

public void actionPerformed(événement ActionEvent) {
    si (event.getSource() == setBPMButton) {
        int bpm = 90;
        Chaîne bpmText = bpmTextField.getText();
        si (bpmText == null || bpmText.contentEquals("")) {
            bpm = 90;
        } autre {
            bpm = Integer.parseInt(bpmTextField.getText());
        }
        contrôleur.setBPM(bpm);
    } sinon si (event.getSource() == increaseBPMButton) {
        contrôleur.augmenteBPM();
    } sinon si (event.getSource() == diminueBPMButton) {
        contrôleur.diminueBPM();
    }
}

public void updateBPM() {
    int bpm = modèle.getBPM(); si
    (bpm == 0) {
        bpmOutputLabel.setText("hors ligne"); }
    else {
        bpmOutputLabel.setText("BPM actuel : " + model.getBPM()); }
}

public void updateBeat() {
    beatBar.setValue(100);
}
}
```

Le contrôleur

paquet headfirst.designpatterns.combined.djview;

```
interface publique ControllerInterface {
    void démarrer();
    vide arrêter();
    void augmenterBPM();
    void diminuerBPM();
    void setBPM(int bpm);
}
```



Prêt à cuire
Code

```

paquet headfirst.designpatterns.combined.djview;

la classe publique BeatController implémente ControllerInterface {
    Modèle BeatModelInterface ;
    vue DJView ;

    public BeatController(BeatModelInterface modèle) {
        ce.modèle = modèle;
        vue = nouveau DJView(ceci, modèle);
        vue.createView();
        vue.createControls();
        vue.disableStopMenuItem();
        vue.enableStartMenuItem();
        modèle.initialize();
    }

    public void start() {
        modèle.on();
        afficher.disableStartMenuItem();
        afficher.enableStopMenuItem();
    }

    public void stop() {
        modèle.off();
        afficher.disableStopMenuItem();
        afficher.enableStartMenuItem();
    }

    public void increaseBPM() {
        int bpm = modèle.getBPM();
        modèle.setBPM(bpm + 1);
    }

    public void diminutionBPM() {
        int bpm = modèle.getBPM();
        modèle.setBPM(bpm - 1);
    }

    public void setBPM(int bpm) {
        modèle.setBPM(bpm);
    }
}

```

Le modèle du cœur

```
paquet headfirst.designpatterns.combined.djview;

classe publique HeartTestDrive {

    public static void main (String[] args) {
        HeartModel heartModel = new HeartModel(); Modèle ControllerInterface =
        new HeartController (heartModel);
    }
}

paquet headfirst.designpatterns.combined.djview;

interface publique HeartModelInterface {
    int getHeartRate();
    void registerObserver(BeatObserver o); void
    removeObserver(BeatObserver o); void
    registerObserver(BPMObserver o); void
    removeObserver(BPMObserver o);
}

paquet headfirst.designpatterns.combined.djview;

importer java.util.*;

la classe publique HeartModel implémente HeartModelInterface, Runnable {
    Liste<BeatObserver> beatObservers = new ArrayList<BeatObserver>();
    Liste<BPMObserver> bpmObservers = new ArrayList<BPMObserver>(); int time =
    1000;
    int bpm = 90;
    Aléatoire aléatoire = nouveau Aléatoire(System.currentTimeMillis());
    Thread thread;

    public HeartModel() {
        thread = nouveau Thread(ceci);
        thread.start();
    }

    public void run() {
        int lastrate = -1;

        pour(;;) {
            int change = random.nextInt(10); si
            (random.nextInt(2) == 0) {
                changement = 0 - changement;
            }
            int rate = 60000/(temps + changement);
        }
    }
}
```

```

        si (taux < 120 && taux > 50) {
            temps += changement;
            notifierBeatObservers();
            si (taux != lastrate) {
                lastrate = taux;
                notifierBPMObservers();
            }
        }
        essayer {
            Thread.sleep(temps);
        } attraper (Exception e) {}
    }
}

public int getHeartRate() {
    retourner 60000/fois ;
}

public void registerObserver(BeatObserver o) {
    beatObservers.add(o);
}

public void removeObserver(BeatObserver o) {
    int i = beatObservers.indexOf(o); si (i >= 0)
    {
        beatObservers.remove(i);
    }
}

public void notifierBeatObservers() {
    pour(int i = 0; i < beatObservers.size(); i++) {
        observer.updateBeat();
    }
}

public void registerObserver(BPMObserver o) {
    bpmObservers.add(o);
}

public void removeObserver(BPMObserver o) {
    int i = bpmObservers.indexOf(o); si (i >=
    0) {
        bpmObservers.remove(i);
    }
}

public void notifierBPMObservers() {
    pour(int i = 0; i < bpmObservers.size(); i++) {
        observer.updateBPM();
    }
}
}

```



Prêt à cuire
Code

L'adaptateur du cœur

```
paquet headfirst.designpatterns.combined.djview;

la classe publique HeartAdapter implémente BeatModelInterface {
    HeartModelInterface coeur;

    public HeartAdapter(HeartModelInterface coeur) {
        ceci.coeur = coeur;
    }

    public void initialize() {}

    public void on() {}

    public void off() {}

    public int getBPM() {
        retourner coeur.getHeartRate();
    }

    public void setBPM(int bpm) {}

    public void registerObserver(BeatObserver o) {
        coeur.registerObserver(o);
    }

    public void removeObserver(BeatObserver o) {
        coeur.removeObserver(o);
    }

    public void registerObserver(BPMObserver o) {
        coeur.registerObserver(o);
    }

    public void removeObserver(BPMObserver o) {
        coeur.removeObserver(o);
    }
}
```

Le contrôleur

```
paquet headfirst.designpatterns.combined.djview;
```



```
la classe publique HeartController implémente ControllerInterface {
```

```
    Modèle HeartModelInterface ;
```

```
    Vue DJView ;
```

```
public HeartController (modèle HeartModelInterface) {
```

```
    ce.modèle = modèle;
```

```
    vue = nouveau DJView(ceci, nouveau HeartAdapter(modèle));
```

```
    vue.createView();
```

```
    vue.createControls();
```

```
    afficher.disableStopMenuItem();
```

```
    afficher.disableStartMenuItem();
```

```
}
```

```
public void start() {}
```

```
public void stop() {}
```

```
public void increaseBPM() {}
```

```
public void diminutionBPM() {}
```

```
public void setBPM(int bpm) {}
```

```
}
```


13mieux vivre avec des motifs

Les modèles dans le Monde réel



Ahhhh, maintenant vous êtes prêt pour un nouveau monde lumineux rempli de

Modèles de conception.Mais avant de vous lancer dans l'ouverture de toutes ces nouvelles portes d'opportunités, nous devons aborder quelques détails que vous rencontrerez dans le monde réel. C'est vrai, les choses deviennent un peu plus complexes qu'ici à Objectville. Venez, nous avons un guide pratique pour vous aider à traverser la transition sur la page suivante...



Le guide d'Objectville pour

Mieux vivre grâce aux modèles de conception

Veuillez accepter notre guide pratique contenant des conseils et astuces pour vivre avec des schémas dans le monde réel. Dans ce guide, vous allez :

-) Découvrez les idées fausses trop courantes sur la définition d'un « modèle de conception ».
-) Découvrez ces catalogues de modèles de conception astucieux et pourquoi vous devez absolument vous en procurer un.
-) Évitez l'embarras d'utiliser un modèle de conception au mauvais moment.
-) Apprenez à conserver les modèles dans les classifications où ils appartiennent.
-) Découvrez que la découverte de modèles n'est pas réservée uniquement aux gourous ; lisez notre guide pratique et devenez vous aussi un rédacteur de modèles.
-) Soyez présent lorsque la véritable identité du mystérieux Gang des Quatre sera révélée.
-) Restez au courant des voisins : les livres de table basse contiennent tous les modèles que l'utilisateur doit posséder.
-) Apprenez à entraîner votre esprit comme un maître zen.
-) Gagnez des amis et influencez les développeurs en améliorant votre vocabulaire de modèles.

Modèle de conception défini

Nous parions que vous avez une assez bonne idée de ce qu'est un modèle après avoir lu ce livre. Mais nous n'avons jamais vraiment donné de définition d'un modèle de conception. Eh bien, vous pourriez être un peu surpris par la définition qui est couramment utilisée :

Un modèleest une solution à un problème dans un contexte.

Ce n'est pas la définition la plus révélatrice, n'est-ce pas ? Mais ne vous inquiétez pas, nous allons parcourir chacune de ces parties : contexte, problème et solution :

Le contexteest la situation dans laquelle le modèle s'applique. Il doit s'agir d'une situation récurrente.

Le problèmefait référence à l'objectif que vous essayez d'atteindre dans ce contexte, mais il fait également référence à toutes les contraintes qui surviennent dans ce contexte.

La solutionc'est ce que vous recherchez : une conception générale que tout le monde peut appliquer et qui résout l'objectif et l'ensemble des contraintes.

Exemple : Vous avez une collection d'objets.

Vous devez traverser les objets sans exposer la collection mise en œuvre.

Encapsuler le itération dans un classe séparée.

C'est une de ces définitions qui prennent un certain temps à assimiler, mais il faut y aller étape par étape. Essayez d'y penser comme ceci :

« Si vous trouvez dans un *contexte* avec un *problème* qui a un objectif qui est affecté par un ensemble de contraintes, vous pouvez alors appliquer une conception qui résout l'objectif et les contraintes et conduit à une *solution*. »

Maintenant, cela semble être beaucoup de travail juste pour comprendre ce qu'est un modèle de conception. Après tout, vous savez déjà qu'un modèle de conception vous donne une solution à un problème de conception récurrent courant. Qu'est-ce que toute cette formalité vous apporte ? Eh bien, vous allez voir qu'en ayant une manière formelle de décrire les modèles, nous pouvons créer un catalogue de modèles, ce qui présente toutes sortes d'avantages.



J'ai réfléchi à la définition en trois parties, et je ne pense pas qu'elle définisse une modèle du tout.

You avez peut-être raison, réfléchissons un peu à cela... Nous avons besoin d'un *problème*, un *solution*, et un *contexte*.

Problème: Comment puis-je arriver au travail à l'heure ?

Contexte: J'ai enfermé mes clés dans la voiture.

Solution: Brisez la vitre, montez dans la voiture, démarrez le moteur et conduisez jusqu'au travail.

Nous avons tous les éléments de la définition : nous avons un problème, qui comprend l'objectif de se rendre au travail, et les contraintes de temps, de distance et probablement d'autres facteurs. Nous avons également un contexte dans lequel les clés de la voiture sont inaccessibles. Et nous avons une solution qui nous permet d'accéder aux clés et de résoudre à la fois les contraintes de temps et de distance. Nous devons avoir un modèle maintenant ! N'est-ce pas ?



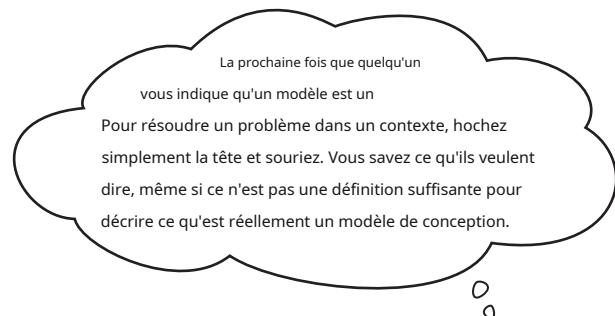
Nous avons suivi la définition du modèle de conception et défini un problème, un contexte et une solution (qui fonctionne !). Est-ce un modèle ? Si ce n'est pas le cas, comment a-t-il échoué ? Pourrions-nous échouer de la même manière lors de la définition d'un modèle de conception OO ?

Regardons de plus près la définition du modèle de conception

Notre exemple semble correspondre à la définition du modèle de conception, mais il ne s'agit pas d'un véritable modèle. Pourquoi ? Pour commencer, nous savons qu'un modèle doit s'appliquer à un problème récurrent. Même si une personne distraite peut souvent laisser ses clés dans la voiture, casser la vitre de la voiture ne constitue pas une solution qui peut être appliquée à plusieurs reprises (ou du moins, ce n'est pas une solution qui peut être appliquée à plusieurs reprises si nous équilibrions l'objectif avec une autre contrainte : le coût).

Cela échoue également sur plusieurs autres points : tout d'abord, il n'est pas facile de prendre cette description, de la transmettre à quelqu'un et de lui demander de l'appliquer à son propre problème. Ensuite, nous avons violé un aspect important mais simple d'un modèle : nous ne lui avons même pas donné de nom ! Sans nom, le modèle ne fait pas partie d'un vocabulaire qui peut être partagé avec d'autres développeurs.

Heureusement, les modèles ne sont pas décrits et documentés comme un simple problème, un contexte et une solution ; nous avons de bien meilleures façons de décrire les modèles et de les rassembler dans un seul ensemble. *catalogues de patrons.*



*there are no
Dumb Questions*

Q: Vais-je voir des descriptions de modèles qui sont présentées comme un problème, un contexte et une solution ?

UN:
Les descriptions de modèles, que vous trouverez généralement dans les catalogues de modèles, sont généralement un peu plus révélatrices que cela. Nous allons examiner les catalogues de modèles en détail dans une minute. Ils décrivent bien plus en détail l'intention et la motivation d'un modèle et les domaines dans lesquels il peut s'appliquer, ainsi que la conception de la solution et les conséquences (bonnes et mauvaises) de son utilisation.

Q: Puis-je modifier légèrement la structure d'un motif pour l'adapter à ma conception ? Ou vais-je devoir m'en tenir à la définition stricte ?

UN:
Bien sûr vous pouvez le modifier. Tout comme les principes de conception, les modèles ne sont pas censés être des lois ou des règles ; ce sont des lignes directrices que vous pouvez modifier pour les adapter à vos besoins. Comme vous l'avez vu, de nombreux exemples concrets ne correspondent pas aux modèles de conception classiques.

Cependant, lorsque vous adaptez des modèles, il n'est jamais inutile de documenter en quoi votre modèle diffère de la conception classique. De cette façon, d'autres développeurs peuvent rapidement reconnaître les modèles que vous utilisez et les différences entre votre modèle et le modèle classique.

Q: Où puis-je obtenir un catalogue de modèles ?

UN:
Le premier et le plus définitif catalogue de modèles est *Modèles de conception : éléments d'un logiciel orienté objet réutilisable*, par Gamma, Helm, Johnson et Vlissides (Addison Wesley). Ce catalogue présente 23 modèles fondamentaux. Nous parlerons un peu plus de ce livre dans quelques pages.

De nombreux autres catalogues de modèles commencent à être publiés dans divers domaines tels que les logiciels d'entreprise, les systèmes simultanés et les systèmes commerciaux.



Les geeks

Que la force soit avec toi

Le modèle de conception

la définition nous dit que le *problème* se

compose d'un *but* et un *ensemble de*

contraintes Les gourous des modèles ont un

terme pour cela : ils les appellent des forces.

Pourquoi ? Eh bien, nous sommes sûrs qu'ils

ont leurs propres raisons, mais si vous vous

souvenez du film, la force « façonne et contrôle

l'Univers ». De même, les forces dans la définition

du modèle façonnent et contrôlent la solution. Ce

n'est que lorsqu'une solution équilibre les deux

côtés de la force (le côté lumineux : votre objectif et le

côté obscur : les contraintes) que nous avons un

modèle utile.

Cette terminologie de « force » peut être assez déroutante lorsque vous la voyez pour la première fois dans les discussions sur les modèles, mais n'oubliez pas qu'il existe deux côtés de la force (objectifs et contraintes) et qu'ils doivent être équilibrés ou résolus pour créer une solution de modèle. Ne laissez pas le jargon vous gêner et que la force soit avec vous !

J'aurais aimé connaître les catalogues de patrons il y a longtemps...



Franc: Raconte-nous tout, Jim. J'ai appris des modèles en lisant quelques articles ici et là.

Jim : Bien sûr, chaque catalogue de modèles prend un ensemble de modèles et décrit chacun d'eux en détail ainsi que sa relation avec les autres modèles.

Joe : Dites-vous qu'il existe plus d'un catalogue de modèles ?

Jim : Bien sûr, il existe des catalogues pour les modèles de conception fondamentaux et il existe également des catalogues sur les modèles spécifiques à un domaine, comme les modèles d'entreprise ou de calcul distribué.

Franc: Quel catalogue consultez-vous ?

Jim : Il s'agit du catalogue GoF classique ; il contient 23 modèles de conception fondamentaux.

Franc: Allez-y !

Jim : C'est vrai, ça veut dire le Gang des Quatre. Le Gang des Quatre, ce sont les gars qui ont mis au point le premier catalogue de patrons.

Joe : Qu'y a-t-il dans le catalogue ?

Jim : Il existe un ensemble de modèles associés. Pour chaque modèle, il existe une description qui suit un modèle et précise de nombreux détails du modèle. Par exemple, chaque modèle a un nom.

Franc:Wow, c'est bouleversant, un nom ! Imaginez ça.

Jim:Attends, Frank. En fait, le nom est très important. Lorsque nous avons un nom pour un motif, cela nous donne un moyen d'en parler ; vous savez, tout ce vocabulaire partagé.

Franc: Ok, ok. Je plaisantais. Allez, qu'est-ce qu'il y a d'autre ?

Jim:Eh bien, comme je le disais, chaque modèle suit un modèle. Pour chaque modèle, nous avons un nom et quelques sections qui nous en disent plus sur le modèle. Par exemple, il y a une section Intention qui décrit ce qu'est le modèle, un peu comme une définition. Ensuite, il y a les sections Motivation et Applicabilité qui décrivent quand et où le modèle peut être utilisé.

Joe:Qu'en est-il du design lui-même ?

Jim: Il existe plusieurs sections qui décrivent la conception de la classe ainsi que toutes les classes qui la composent et leurs rôles. Il existe également une section qui décrit comment implémenter le modèle et souvent un exemple de code pour vous montrer comment procéder.

Franc:On dirait qu'ils ont pensé à tout.

Jim:Il y a plus. Il y a aussi des exemples où le modèle a été utilisé dans des systèmes réels, ainsi que ce que je pense être l'une des sections les plus utiles : la relation entre le modèle et d'autres modèles.

Franc:Oh, tu veux dire qu'ils te disent des choses comme la façon dont les modèles d'État et de stratégie diffèrent ?

Jim: Exactement!

Joe:Alors Jim, comment utilises-tu le catalogue ? Lorsque tu as un problème, est-ce que tu fouilles dans le catalogue pour trouver une solution ?

Jim:J'essaie d'abord de me familiariser avec tous les modèles et leurs relations. Ensuite, lorsque j'ai besoin d'un modèle, j'ai une idée de ce que c'est. Je reviens en arrière et je regarde les sections Motivation et Applicabilité pour m'assurer que je l'ai bien compris. Il y a aussi une autre section très importante : Conséquences. Je la passe en revue pour m'assurer qu'il n'y aura pas d'effet imprévu sur ma conception.

Franc:Cela a du sens. Alors, une fois que vous savez que le modèle est bon, comment l'intégrer à votre conception et le mettre en œuvre ?

Jim:C'est là qu'intervient le diagramme de classes. Je lis d'abord la section Structure pour examiner le diagramme, puis la section Participants pour m'assurer que je comprends le rôle de chaque classe. À partir de là, je l'intègre dans ma conception, en apportant les modifications nécessaires pour l'adapter. Ensuite, je consulte les sections Implémentation et Exemple de code pour m'assurer que je connais les bonnes techniques d'implémentation ou les pièges que je pourrais rencontrer.

Joe:Je vois comment un catalogue va vraiment accélérer mon utilisation des modèles !

Franc:Totalement. Jim, peux-tu nous décrire un modèle ?

Tous les modèles d'un catalogue commencent par un nom. Le nom est un élément essentiel d'un modèle : sans un bon nom, un modèle ne peut pas faire partie du vocabulaire que vous partagez avec d'autres développeurs.

La motivation vous donne un scénario concret qui décrit le problème et comment la solution résout le problème.

L'applicabilité décrit les situations dans lesquelles le modèle peut être appliquée.

Les participants sont les classes et les objets de la conception. Cette section décrit leurs responsabilités et leurs rôles dans le modèle.

Les conséquences décrivent les effets que l'utilisation de ce modèle peut avoir : bons et mauvais.

L'implémentation fournit les techniques que vous devez utiliser lors de la mise en œuvre de ce modèle, ainsi que les problèmes auxquels vous devez faire attention.

Utilisations connues décrit exemples de ce modèle trouvés dans des systèmes réels.

SINGLETON

Création d'objet

Intention

Et aliquid, vellesto ent lora feus ad illas rperci tis, quat non sequam il ea at nim nos do enim qui eratissim ex ea faci tis. Sequi dion utat, volore magnisi.

Motivation

Et aliquis, vellesto ent lora feus ad illas rperci tis, quat non sequam il ea at nim nos do enim qui dionsequi dighib nūmby nābi esquet. Dūs nūluprem ipsam esete consulut wissi.

Structure



Participants

Dūs nūluprem ipsam esete consulut wissi. Et magna aliqui blādet, consuillandre dolore magna feus nos alt ad magnim quāte modolore venti lut luptat prat. Dui blāore min ea feupit ing ent laore magnibh eniat wissi. Sussilla ad minicini blām dolorse rclit int. nūlullu tpatiss ismodignibh er.

- ❖ Ad dolore dolore et, verci ens ent ip elesequiut ad esectem ing ea con eros autem diam
- A feus nos alt ad magnim quāte modolore venti lut luptat prat. Dui blāore min ea feupit ing ent laore magnibh eniat wissi.
- Ad magnim quāte modolore venti lut luptat prat. Dui blāore min ea feupit ing ent

Collaborations

❖ Feupit ing ent laore magnibh eniat wissi. Et, sussilla ad minicini blām dolorse rclit int.

1. Dolore dolore et, verci ens ent ip elesequiut ad esectem ing ea con eros autem diam nūlullu tpatiss ismodignibh er.

2. Modolore venti lut luptat prat. Dui blāore min ea feupit ing ent laore magnibh eniat verci ens ent ip elesequiut ad esectem.

3. Dolore dolore et, verci ens ent ip elesequiut ad esectem ing ea con eros autem diam nūlullu tpatiss ismodignibh er.

4. Modolore venti lut luptat prat. Dui blāore min ea feupit ing ent laore magnibh eniat wissi. Et, sussilla ad minicini blām dolorse rclit int, conse dolore dolore et, verci ens ent ip elesequiut ad esectem.

Implémentation/Exemple de code

Dūs nūluprem ipsam esete consulut wissi. Et magna aliqui blādet, consuillandre dolore magna feus nos alt ad magnim quāte modolore venti lut luptat prat. Dui blāore min ea feupit ing ent laore magnibh eniat wissi. Sussilla ad minicini blām dolorse rclit int, conse dolore dolore et, verci ens ent ip elesequiut ad esectem ing ea con eros autem diam nūlullu tpatiss ismodignibh er.

```

class public Singleton {
    Singleton static priv uniqueInstance;
    // autres variables d'instance utiles ici
    Singleton priv();
    public statique synchronisé Singleton getInstance()
    {
        si (uniqueInstance == null) {
            uniqueInstance = nouveau Singleton();
        }
        renvoyer une instance unique ;
    }
    // autres méthodes utiles ici
}
  
```

Nos alt ad magnim quāte modolore venti lut luptat prat. Dui blāore min ea feupit ing ent laore magnibh eniat wissi. Sussilla ad minicini blām dolorse rclit int, conse dolore dolore et, verci ens ent ip elesequiut ad esectem ing ea con eros autem diam nūlullu tpatiss ismodignibh er.

Utilisations connues

Dūs nūluprem ipsam esete consulut wissi. Et magna aliqui blādet, consuillandre dolore magna feus nos alt ad magnim quāte modolore venti lut luptat prat. Dui blāore min ea feupit ing ent laore magnibh eniat wissi. Sussilla ad minicini blām dolorse rclit int, conse dolore dolore et, verci ens ent ip elesequiut ad esectem.

Dūs nūluprem ipsam esete consulut wissi. Et magna aliqui blādet, consuillandre dolore magna feus nos alt ad magnim quāte modolore venti lut luptat prat. Dui blāore min ea feupit ing ent laore magnibh eniat wissi. Sussilla ad minicini blām dolorse rclit int, conse dolore dolore et, verci ens ent ip elesequiut ad esectem.

Modèles associés

Elesequiut ad esectem ing ea con eros autem diam nūlullu tpatiss ismodignibh er, alt ad magnim quāte modolore venti lut luptat prat. Dui blāore min ea feupit ing ent laore magnibh eniat wissi. Sussilla ad minicini blām dolorse rclit int, conse dolore dolore et, verci ens ent ip elesequiut ad esectem.

Il s'agit de la classification du modèle ou catégorie. Nous parlerons à ce sujet dans quelques pages.

L'intention décrit ce que fait le modèle dans une courte déclaration. Vous pouvez également considérer cela comme la définition du modèle (tout comme nous l'avons utilisé dans ce livre).

La structure fournit un diagramme illustrant les relations entre les classes qui participent au modèle.

Les collaborations nous indiquent comment les participants travaillent ensemble dans le modèle.

Exemple de code fournit le code fragments qui pourrait aider à votre mise en œuvre.

Modèles associés décrit le relation entre ce modèle et d'autres.

there are no Dumb Questions

Q: Est-il possible de créer ses propres modèles de conception ? Ou faut-il être un « gourou des modèles » pour le faire ?

UN: Tout d'abord, n'oubliez pas que les modèles sont découverts et non créés. Ainsi, n'importe qui peut découvrir un modèle de conception et ensuite rédiger sa description. Cependant, ce n'est pas facile et cela ne se produit pas rapidement, ni souvent. Être un « rédacteur de modèles » demande de l'engagement.

Vous devez d'abord réfléchir à la raison pour laquelle vous souhaitez le faire : la majorité des gens ne créent pas de modèles ; ils les utilisent simplement. Cependant, vous travaillez peut-être dans un domaine spécialisé pour lequel vous pensez que de nouveaux modèles seraient utiles, ou vous avez peut-être trouvé une solution à ce que vous pensez être un problème récurrent, ou vous souhaitez peut-être simplement vous impliquer dans la communauté des modèles et contribuer au corpus croissant de travaux.

Q: Je suis partant ; comment puis-je commencer ?

UN: Comme pour toute discipline, plus vous en savez, mieux c'est. Il est essentiel d'étudier les modèles existants, leur fonction et la manière dont ils se rapportent aux autres modèles. Cela vous permet non seulement de vous familiariser avec la manière dont les modèles sont créés, mais aussi d'éviter de réinventer la roue. À partir de là, vous devrez commencer à écrire vos modèles sur papier, afin de pouvoir les communiquer à d'autres développeurs. Nous allons parler plus en détail de la manière de communiquer vos modèles dans un instant. Si vous êtes vraiment intéressé, vous voudrez lire la section qui suit ces questions-réponses.

Q: Comment savoir si j'ai vraiment un modèle ?

UN: C'est une très bonne question : vous n'avez pas de modèle tant que d'autres ne l'ont pas utilisé et n'ont pas constaté qu'il fonctionnait. En général, vous n'avez pas de modèle tant qu'il ne satisfait pas à la « règle des trois ». Cette règle stipule qu'un modèle ne peut être qualifié de modèle que s'il a été appliqué dans une solution concrète au moins trois fois.

Alors vous voulez devenir une star des modèles de conception ?

Eh bien, écoutez maintenant ce que je dis.

Procurez-vous un catalogue de modèles,

Alors prenez le temps de bien l'apprendre.

Et quand vous avez bien décrit votre description,

Et trois développeurs sont d'accord sans se battre,

Alors vous saurez que c'est un modèle, c'est sûr.



Sur l'air de « Alors tu veux être une star du rock'n'roll ».

Alors, vous voulez devenir rédacteur de modèles de conception

Fais tes devoirs. Vous devez bien connaître les modèles existants avant de pouvoir en créer un nouveau. La plupart des modèles qui semblent nouveaux ne sont en fait que des variantes de modèles existants. En étudiant les modèles, vous apprenez à mieux les reconnaître et à les relier à d'autres modèles.

Prenez le temps de réfléchir, d'évaluer. Votre expérience, les problèmes que vous avez rencontrés et les solutions que vous avez utilisées, sont à l'origine des idées de modèles. Prenez donc le temps de réfléchir à vos expériences et de les éplucher pour trouver des modèles nouveaux qui reviennent régulièrement. N'oubliez pas que la plupart des modèles sont des variantes de modèles existants et non de nouveaux modèles. Et lorsque vous trouvez ce qui ressemble à un nouveau modèle, son applicabilité peut être trop limitée pour être qualifiée de véritable modèle.

Écrivez vos idées sur papier de manière à ce que les autres puissent les comprendre. La recherche de nouveaux modèles n'est pas d'une grande utilité si d'autres ne peuvent pas utiliser votre découverte. Vous devez documenter vos modèles candidats afin que d'autres puissent les lire, les comprendre et les appliquer à leur propre solution, puis vous fournir des commentaires. Heureusement, vous n'avez pas besoin d'inventer votre propre méthode de documentation de vos modèles. Comme vous l'avez déjà vu avec le modèle GoF, beaucoup de réflexion a déjà été consacrée à la manière de décrire les modèles et leurs caractéristiques.

Demandez à d'autres d'essayer vos modèles, puis affinez-les encore et encore. Ne vous attendez pas à obtenir un modèle parfait du premier coup. Considérez votre modèle comme un travail en cours qui s'améliorera au fil du temps. Demandez à d'autres développeurs de revoir votre modèle candidat, de l'essayer et de vous donner leur avis. Intégrer ces commentaires dans votre description et réessayez. Votre description ne sera jamais parfaite, mais à un moment donné, elle devrait être suffisamment solide pour que d'autres développeurs puissent la lire et la comprendre.

N'oubliez pas la règle des trois. N'oubliez pas que si votre modèle n'a pas été appliqué avec succès dans trois solutions concrètes, il ne peut pas être considéré comme un modèle. C'est une autre bonne raison de mettre votre modèle entre les mains d'autres personnes afin qu'elles puissent l'essayer, vous donner leur avis et vous permettre de converger vers un modèle fonctionnel.

Utilisez l'un des modèles de modèles existants pour Définissez votre modèle. Ces modèles ont fait l'objet d'une réflexion approfondie et d'autres utilisateurs de modèles reconnaîtront le format.





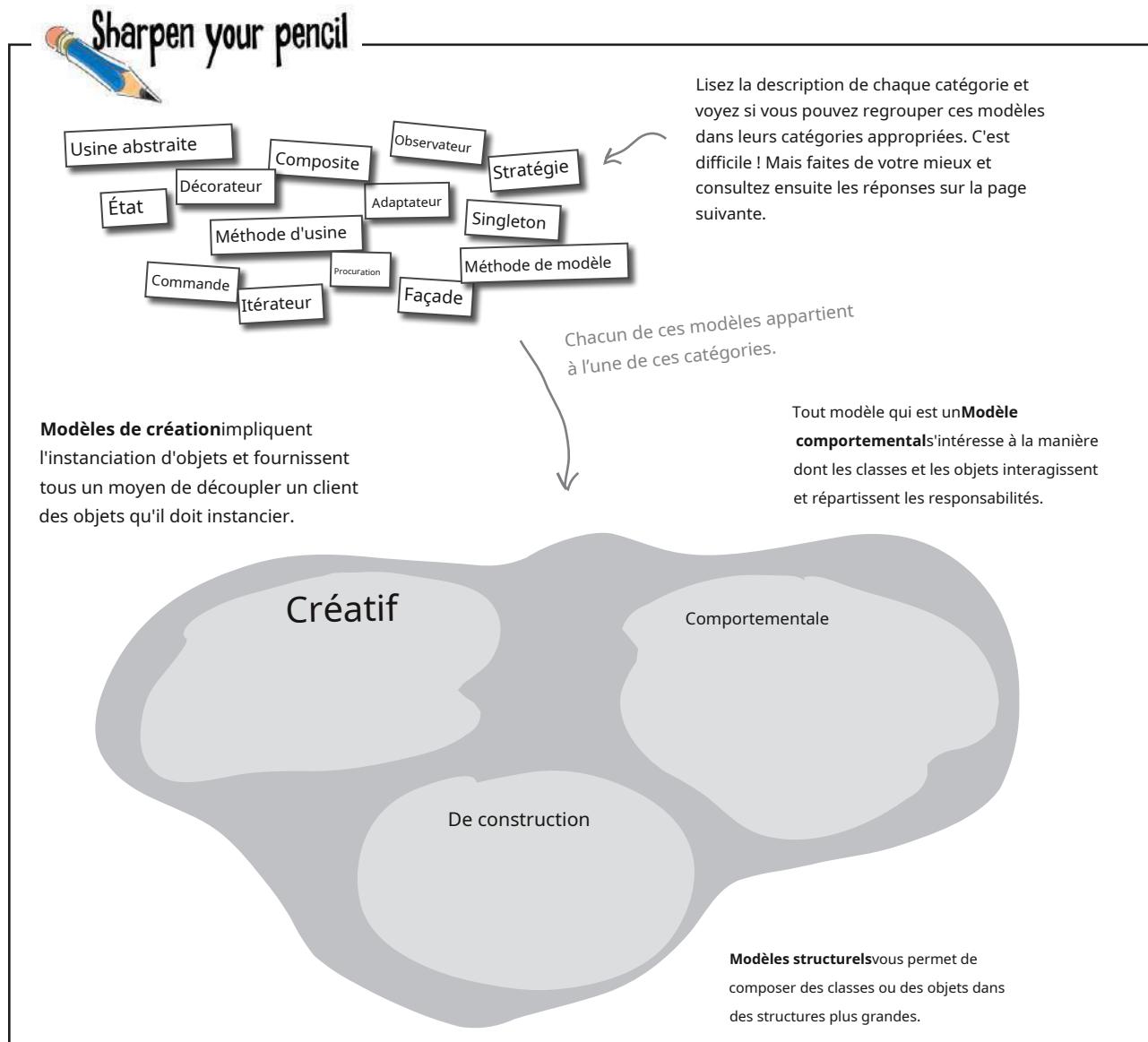
Associez chaque motif à sa description :

Modèle	Description
Décorateur	Enveloppe un objet et lui fournit une interface différente.
État	Les sous-classes décident comment implémenter les étapes d'un algorithme.
Itérateur	Les sous-classes décident quelles classes concrètes créer.
Façade	Garantit qu'un seul et unique objet est créé.
Stratégie	Encapsule les comportements interchangeables et utilise la délégation pour décider lequel utiliser.
Procuration	Les clients traitent les collections d'objets et les objets individuels de manière uniforme.
Méthode d'usine	Encapsule les comportements basés sur l'état et utilise la délégation pour basculer entre les comportements.
Adaptateur	Fournit un moyen de parcourir une collection d'objets sans exposer son implémentation.
Observateur	Simplifie l'interface d'un ensemble de classes. Encapsule un objet pour fournir un nouveau comportement.
Méthode de modèle	Permet à un client de créer des familles d'objets sans spécifier leurs classes concrètes.
Composite	Permet aux objets d'être notifiés lorsque leur état change.
Singleton	Encapsule un objet pour contrôler l'accès à celui-ci.
Usine abstraite	Encapsule une requête sous forme d'objet.
Commande	

Organisation des modèles de conception

À mesure que le nombre de modèles de conception découverts augmente, il est logique de les partitionner en classifications afin que nous puissions les organiser, affiner nos recherches à un sous-ensemble de tous les modèles de conception et effectuer des comparaisons au sein d'un groupe de modèles.

Dans la plupart des catalogues, vous trouverez des modèles regroupés dans l'un des quelques systèmes de classification. Le système le plus connu était celui utilisé par le premier catalogue de modèles et répartissait les modèles en trois catégories distinctes en fonction de leurs objectifs : création, comportement et structurel.

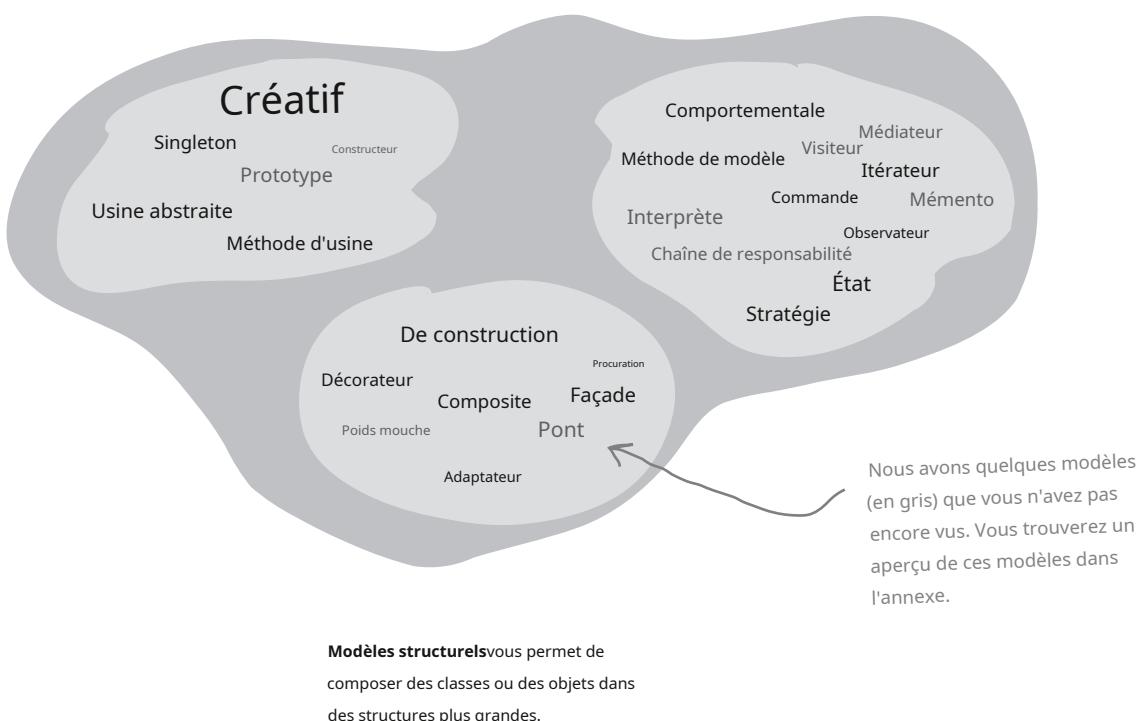


Catégories de modèles

Voici le regroupement des motifs en catégories. Vous avez probablement trouvé l'exercice difficile, car de nombreux motifs semblent pouvoir être classés dans plusieurs catégories. Ne vous inquiétez pas, tout le monde a du mal à déterminer les bonnes catégories pour les motifs.

Modèles de création impliquent l'instanciation d'objets et fournissent tous un moyen de découpler un client des objets qu'il doit instancier.

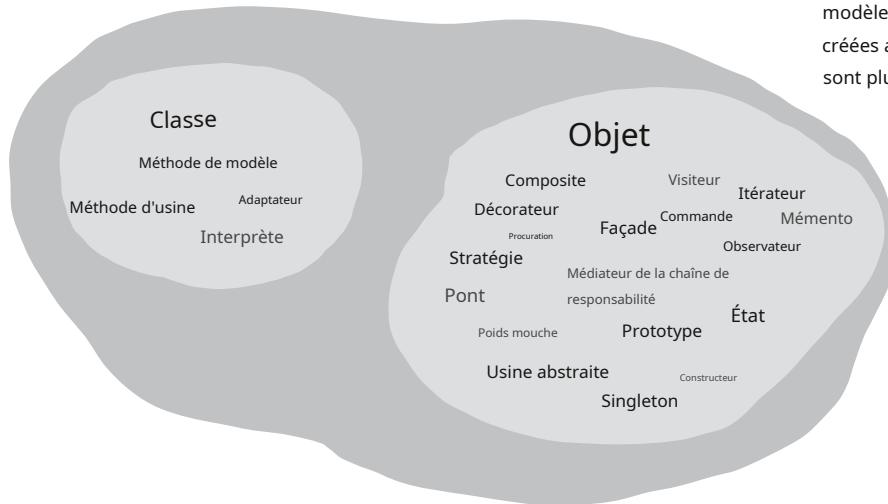
Tout modèle qui est un **Modèle comportemental** s'intéresse à la manière dont les classes et les objets interagissent et répartissent les responsabilités.



Les modèles sont souvent classés par un deuxième attribut : si le modèle traite ou non de classes ou d'objets :

Modèles de classe Décrire comment les relations entre les classes sont définies via l'héritage. Les relations dans les modèles de classe sont établies au moment de la compilation.

Modèles d'objets décrivent les relations entre les objets et sont principalement définies par la composition. Les relations dans les modèles d'objets sont généralement créées au moment de l'exécution et sont plus dynamiques et flexibles.



Notez qu'il y a beaucoup plus de modèles d'objets que modèles de classe !

there are no Dumb Questions

Q: Sont-ce les seuls systèmes de classification ?

UN: Non, d'autres schémas ont été proposés. Certains d'entre eux commencent par les trois catégories, puis ajoutent des sous-catégories, comme « Modèles de découplage ». Vous devez vous familiariser avec les schémas les plus courants pour organiser les modèles, mais n'hésitez pas à créer les vôtres, si cela vous aide à mieux comprendre les modèles.

Q: Est-ce que l'organisation des modèles en catégories vous aide vraiment à vous en souvenir ?

UN: Cela vous donne certainement un cadre de comparaison. Mais beaucoup de gens sont déroutés par les catégories de création, de structure et de comportement ; souvent, un modèle semble appartenir à plusieurs catégories. Le plus important est de connaître les modèles et les relations entre eux. Lorsque les catégories vous aident, utilisez-les !

Q: Pourquoi le modèle Decorator est-il dans la catégorie structurelle ? J'aurais pensé qu'il s'agissait d'un modèle comportemental ; après tout, il ajoute du comportement !

UN: Oui beaucoup de développeurs disent ça ! Voici la réflexion derrière la classification Gang of Four : les modèles structurels décrivent comment les classes et les objets sont composés pour créer de nouvelles structures ou de nouvelles fonctionnalités. Le modèle Decorator vous permet de composer des objets en enveloppant un objet avec un autre pour fournir de nouvelles fonctionnalités. L'accent est donc mis sur la façon dont vous composez les objets de manière dynamique pour obtenir des fonctionnalités, plutôt que sur la communication et l'interconnexion entre les objets, ce qui est le but des modèles comportementaux. Mais n'oubliez pas que l'intention de ces modèles est différente, et c'est souvent la clé pour comprendre à quelle catégorie appartient un modèle.



Gourou et étudiant...

Gourou: Étudiant, vous avez l'air troublé.

Étudiant: Oui, je viens d'apprendre la classification des motifs et je suis confus.

Gourou: Continuer...

Étudiant: Après avoir beaucoup appris sur les modèles, on vient de me dire que chaque modèle correspond à l'une des trois classifications suivantes : structurel, comportemental ou créatif. Pourquoi avons-nous besoin de ces classifications ?

Gourou: Chaque fois que nous avons une grande collection de quelque chose, nous trouvons naturellement des catégories dans lesquelles classer ces éléments. Cela nous aide à penser aux éléments à un niveau plus abstrait.

Étudiant: Gourou, pouvez-vous me donner un exemple ?

Gourou: Bien sûr. Prenons l'exemple des automobiles : il existe de nombreux modèles différents et nous les classons naturellement dans des catégories telles que les voitures économiques, les voitures de sport, les SUV, les camions et les voitures de luxe.

Gourou: Vous avez l'air choqué ; cela n'a-t-il pas du sens ?

Étudiant: Guru, cela a beaucoup de sens, mais je suis choqué que tu en saches autant sur les voitures !

Gourou: Je ne peux pas m'identifier aux fleurs de lotus ou aux bols de riz. Maintenant, puis-je continuer ?

Étudiant: Oui, oui, je suis désolé, continuez s'il vous plaît.

Gourou: Une fois que vous avez établi des classifications ou des catégories, vous pouvez facilement parler des différents groupes : « Si vous faites le trajet en voiture de montagne de la Silicon Valley à Santa Cruz, une voiture de sport avec une bonne tenue de route est la meilleure option. » Ou encore, « Avec la détérioration de la situation pétrolière, vous voulez vraiment acheter une voiture économique ; elles sont plus économies en carburant. »

Étudiant: Ainsi, en ayant des catégories, nous pouvons parler d'un ensemble de modèles en tant que groupe. Nous savons peut-être que nous avons besoin d'un modèle de création, sans savoir exactement lequel, mais nous pouvons quand même parler de modèles de création.

Gourou: Oui, et cela nous permet également de comparer un membre au reste de la catégorie. Par exemple, « La Mini est vraiment la voiture compacte la plus élégante du marché » ou, pour affiner notre recherche, « J'ai besoin d'une voiture économique en carburant ».

Étudiant:Je vois. Je dirais donc que le modèle d'adaptateur est le meilleur modèle structurel pour modifier l'interface d'un objet.

Gourou:Oui. Nous pouvons également utiliser les catégories dans un autre but : se lancer dans un nouveau territoire. Par exemple, « Nous voulons vraiment livrer une voiture de sport avec Ferrari « Des performances à des prix Honda. »

Étudiant:Cela ressemble à un piège mortel. **Gourou:**je suis désolé, je ne vous ai pas entendu, étudiant.

Étudiant:Euh, j'ai dit : « Je vois ça. »

Étudiant:Les catégories nous permettent donc de réfléchir à la manière dont les groupes de modèles sont liés et à la manière dont les modèles au sein d'un groupe sont liés les uns aux autres. Elles nous permettent également d'extrapoler vers de nouveaux modèles. Mais pourquoi existe-t-il trois catégories et non quatre ou cinq ?

Gourou:Ah, comme les étoiles dans le ciel nocturne, il existe autant de catégories que vous voulez voir. Trois est un nombre pratique et un nombre que beaucoup de gens ont décidé de constituer un joli groupe de motifs. Mais d'autres ont suggéré quatre, cinq ou plus.



Penser en termes de modèles

Contextes, contraintes, forces, catalogues, classifications... ça commence à paraître très académique. D'accord, tout ça est important et la connaissance est un pouvoir. Mais, soyons réalistes, si vous comprenez les choses académiques et que vous n'avez pas les connaissances nécessaires, vous pouvez les utiliser pour vous aider. *Expérience et pratique* en utilisant des modèles, cela ne fera pas beaucoup de différence dans votre vie.

Voici un guide rapide pour vous aider à démarrer *penser en termes de modèles*.

Qu'entendons-nous par là ? Nous voulons dire être capable d'observer un design et de voir où les motifs s'intègrent naturellement et où ils ne s'intègrent pas.



Votre cerveau et les modèles

Restez simple (KISS)

Tout d'abord, lorsque vous concevez, résolvez les problèmes de la manière la plus simple possible. Votre objectif doit être la simplicité, et non pas « comment puis-je appliquer un modèle à ce problème ? » Ne pensez pas que vous n'êtes pas un développeur expérimenté si vous n'utilisez pas de modèle pour résoudre un problème. D'autres développeurs apprécieront et admireront la simplicité de votre conception. Cela dit, parfois, la meilleure façon de garder votre conception simple et flexible est d'utiliser un modèle.

Les modèles de conception ne sont pas une solution miracle ; en fait, ils ne sont même pas une solution miracle !

Les modèles, comme vous le savez, sont des solutions générales aux problèmes récurrents. Les modèles ont également l'avantage d'être bien testés par de nombreux développeurs. Ainsi, lorsque vous constatez le besoin d'un modèle, vous pouvez dormir sur vos deux oreilles en sachant que de nombreux développeurs ont déjà été confrontés à ce problème et l'ont résolu en utilisant des techniques similaires.

Cependant, les modèles ne sont pas une solution miracle. Vous ne pouvez pas en brancher un, le compiler, puis déjeuner tôt. Pour utiliser des modèles, vous devez également réfléchir aux conséquences sur le reste de votre conception.

Vous savez que vous avez besoin d'un modèle lorsque...

Ah... la question la plus importante : quand utiliser un modèle ? Lorsque vous abordez votre conception, introduisez un modèle lorsque vous êtes sûr qu'il résout un problème de votre conception. Si une solution plus simple pourrait fonctionner, réfléchissez-y avant de vous engager à utiliser un modèle.

Connaissance C'est là que votre expérience et vos connaissances entrent en jeu. Une fois que vous êtes sûr qu'une solution simple ne répondra pas à vos besoins, vous devez considérer le problème ainsi que l'ensemble des contraintes sous lesquelles la solution devra fonctionner. Ces éléments vous aideront à faire correspondre votre problème à un modèle. Si vous avez une bonne connaissance des modèles, vous connaissez peut-être un modèle qui correspond bien. Sinon, examinez les modèles qui semblent pouvoir résoudre le problème. Les sections sur l'intention et l'applicabilité des catalogues de modèles sont particulièrement utiles à cet effet. Une fois que vous avez trouvé un modèle qui semble correspondre, assurez-vous qu'il a un ensemble de conséquences avec lesquelles vous pouvez vivre et étudiez son effet sur le reste de votre conception. Si tout semble bon, allez-y !

Il existe une situation dans laquelle vous souhaiterez utiliser un modèle même si une solution plus simple fonctionnerait : lorsque vous vous attendez à ce que certains aspects de votre système varient. Comme nous l'avons vu, l'identification des zones de changement dans votre conception est généralement un bon signe qu'un modèle est nécessaire. Assurez-vous simplement d'ajouter des modèles pour gérer ces changements. *changement pratique* cela est susceptible de se produire, non *changement hypothétique*. Cela peut arriver.

Le moment de la conception n'est pas le seul moment où vous devez envisager d'introduire des modèles ; vous devrez également le faire au moment de la refactorisation.

Le temps du refactoring est le temps des patterns !

Refactorisation Il s'agit du processus consistant à apporter des modifications à votre code pour améliorer la manière dont il est organisé. L'objectif est d'améliorer sa structure, et non de modifier son comportement. C'est le moment idéal pour réexaminer votre conception afin de voir si elle pourrait être mieux structurée avec des modèles. Par exemple, un code rempli d'instructions conditionnelles peut signaler la nécessité du modèle d'état. Ou bien, il est peut-être temps de nettoyer les dépendances concrètes avec Factory. Des livres entiers ont été écrits sur le sujet du refactoring avec des modèles, et à mesure que vos compétences se développeront, vous voudrez étudier davantage ce domaine.

Supprimez ce dont vous n'avez pas vraiment besoin. N'ayez pas peur de supprimer un modèle de conception de votre conception.

Personne ne parle jamais du moment où il faut supprimer un motif. On pourrait penser que c'est un blasphème ! Non, nous sommes tous des adultes ici ; nous pouvons le supporter.

Alors, quand faut-il supprimer un modèle ? Lorsque votre système est devenu complexe et que la flexibilité que vous aviez prévue n'est plus nécessaire. En d'autres termes, lorsqu'une solution plus simple sans modèle serait préférable.

Si vous n'en avez pas besoin maintenant, ne le faites pas maintenant.

Les modèles de conception sont puissants et il est facile de voir toutes sortes de façons de les utiliser dans vos conceptions actuelles. Les développeurs aiment naturellement créer de belles architectures prêtes à affronter les changements de toutes les directions.

Résistez à la tentation. Si vous avez un besoin pratique de prendre en charge un changement dans une conception aujourd'hui, n'hésitez pas à utiliser un modèle pour gérer ce changement. Cependant, si la raison n'est qu'hypothétique, n'ajoutez pas le modèle ; il ne fera qu'ajouter de la complexité à votre système, et vous n'en aurez peut-être jamais besoin !

Concentrez votre réflexion sur le design, et non sur les modèles. Utilisez des modèles lorsque vous en avez un besoin naturel. Si quelque chose de plus simple fonctionne, utilisez-le.





ru et étudiant...

russe:Etudiant, votre formation initiale touche à sa fin.

Quels sont vos projets ?

Étudiant:Je vais à Disneyland ! Et ensuite je vais commencer à créer plein de codes avec des motifs !

Gourou:Oh, attends. N'utilise jamais tes gros canons

à moins que vous n'y soyez obligé.

Étudiant:Que voulez-vous dire, Guru ? Maintenant que j'ai appris les modèles de conception, ne devrais-je pas les utiliser dans toutes mes conceptions pour obtenir un maximum de puissance, de flexibilité et de facilité de gestion ?

Gourou:Non, les modèles sont un outil, et un outil qui ne doit être utilisé qu'en cas de besoin. Vous avez également passé beaucoup de temps à apprendre les principes de conception. Commencez toujours par vos principes et créez le code le plus simple possible qui fasse le travail. Cependant, si vous voyez le besoin d'un modèle émerger, utilisez-le.

Étudiant:Je ne devrais donc pas construire mes créations à partir de modèles ?

Gourou:Cela ne devrait pas être votre objectif lorsque vous commencez une conception. Laissez les motifs émerger naturellement au fur et à mesure que votre conception progresse.

Étudiant:Si les modèles sont si géniaux, pourquoi devrais-je être si prudent lorsque je les utilise ?

Gourou:Les modèles peuvent introduire de la complexité, et nous ne voulons jamais de complexité là où elle n'est pas nécessaire. Mais les modèles sont puissants lorsqu'ils sont utilisés là où ils sont nécessaires. Comme vous le savez déjà, les modèles sont une expérience de conception éprouvée qui peut être utilisée pour éviter les erreurs courantes. Ils constituent également un vocabulaire partagé pour communiquer notre conception aux autres.

Étudiant:Eh bien, quand savons-nous qu'il est acceptable d'introduire des modèles de conception ?

Gourou:Introduisez un modèle lorsque vous êtes sûr qu'il est nécessaire de résoudre un problème dans votre conception, ou lorsque vous êtes certain qu'il est nécessaire de faire face à un changement futur dans les exigences de votre application.

Étudiant:Je suppose que mon apprentissage va continuer même si je comprends déjà beaucoup de modèles.

Gourou:Oui, apprendre à gérer la complexité et les changements dans les logiciels est un travail de toute une vie. Mais maintenant que vous connaissez un bon ensemble de modèles, le moment est venu de les appliquer là où cela est nécessaire dans votre conception et de continuer à apprendre d'autres modèles.

Étudiant:Attends une minute, tu veux dire que je ne les connais pas TOUS ?

Gourou:Etudiant, vous avez appris les modèles fondamentaux ; vous allez découvrir qu'il en existe bien d'autres, notamment des modèles qui s'appliquent uniquement à des domaines particuliers tels que les systèmes concurrents et les systèmes d'entreprise. Mais maintenant que vous connaissez les bases, vous êtes en bonne position pour les apprendre.

Votre esprit sur les modèles



Débutant

Esprit

« J'ai besoin d'un modèle pour Hello World. »

Le débutant utilise des modèles partout. C'est une bonne chose : le débutant acquiert beaucoup d'expérience et de pratique avec les modèles. Il pense également : « Plus j'utilise de modèles, meilleur est le design. » Il apprendra que ce n'est pas le cas, que tous les designs doivent être aussi simples que possible. La complexité et les modèles ne doivent être utilisés que là où ils sont nécessaires pour une extensibilité pratique.



Intermédiaire
Esprit

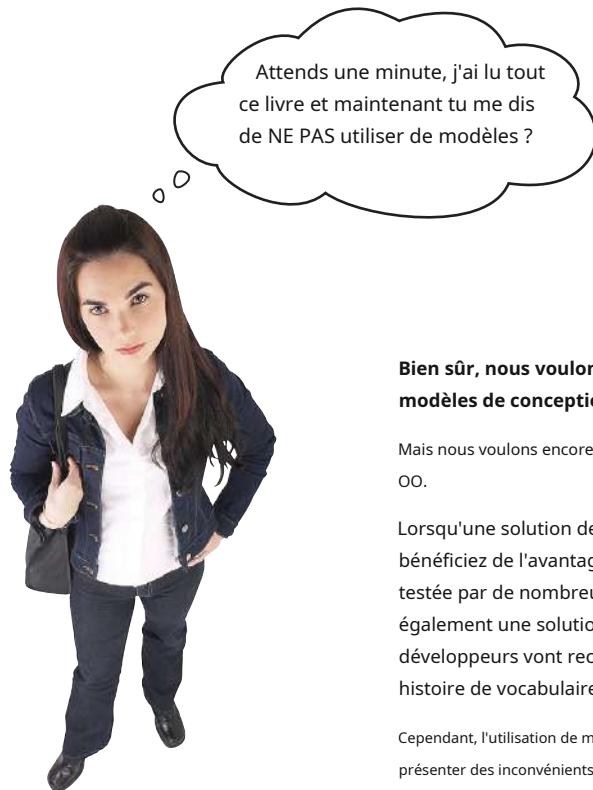
« Peut-être que j'ai besoin d'un Singleton ici. »



Zen Esprit

« C'est un endroit naturel pour Décorateur. »

L'esprit zen est capable de voir des modèles là où ils s'intègrent naturellement. L'esprit zen n'est pas obsédé par l'utilisation de modèles ; il recherche plutôt des solutions simples qui résolvent au mieux le problème. L'esprit zen pense en termes de principes d'objet et de leurs compromis. Lorsqu'un besoin d'un modèle apparaît naturellement, l'esprit zen l'applique en sachant bien qu'il peut nécessiter une adaptation. L'esprit zen voit également les relations avec des modèles similaires et comprend les subtilités des différences dans l'intention des modèles connexes. *L'esprit zen est aussi un esprit débutant—il ne laisse pas toutes ces connaissances sur les modèles influencer de manière excessive les décisions de conception.*



AVERTISSEMENT : l'utilisation excessive de modèles de conception peut conduire à un code carrément sur-conçu. Optez toujours pour la solution la plus simple qui fait le travail et introduisez des modèles là où le besoin s'en fait sentir.

Bien sûr, nous voulons que vous utilisiez des modèles de conception !

Mais nous voulons encore plus que vous soyez un bon concepteur OO.

Lorsqu'une solution de conception nécessite un modèle, vous bénéficiez de l'avantage d'utiliser une solution qui a été testée par de nombreux développeurs. Vous utilisez également une solution bien documentée et que d'autres développeurs vont reconnaître (vous savez, toute cette histoire de vocabulaire partagé).

Cependant, l'utilisation de modèles de conception peut également présenter des inconvénients. Les modèles de conception introduisent souvent des classes et des objets supplémentaires, ce qui peut accroître la complexité de vos conceptions. Les modèles de conception peuvent également ajouter des couches supplémentaires à votre conception, ce qui ajoute non seulement de la complexité, mais aussi de l'inefficacité.

De plus, l'utilisation d'un modèle de conception peut parfois être tout simplement excessive. Souvent, vous pouvez vous appuyer sur vos principes de conception et trouver une solution beaucoup plus simple pour résoudre le même problème. Si cela se produit, ne vous y opposez pas. Utilisez la solution la plus simple.

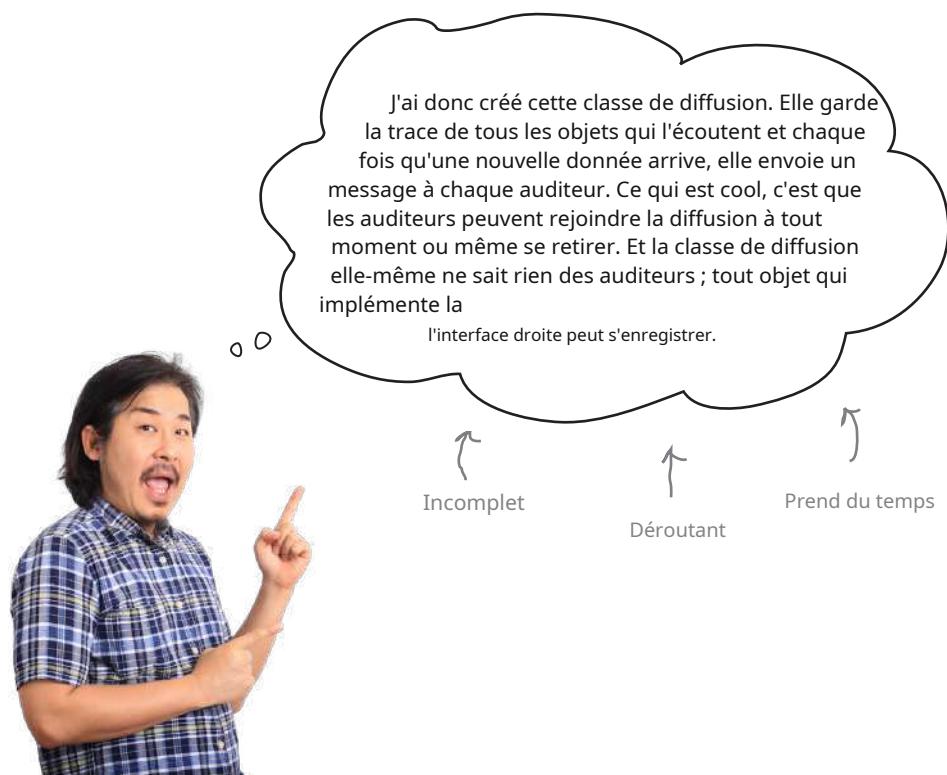
Mais ne vous laissez pas décourager. Lorsqu'un modèle de conception est l'outil adapté à la tâche, les avantages sont nombreux.

N'oubliez pas la puissance du vocabulaire partagé

Nous avons passé tellement de temps dans ce livre à discuter des rouages de l'OO qu'il est facile d'oublier le côté humain des modèles de conception : ils ne vous aident pas seulement à charger votre cerveau de solutions, ils vous donnent également un vocabulaire partagé avec d'autres développeurs. Ne sous-estimez pas la puissance d'un vocabulaire partagé, c'est l'un des/*es plus grands avantages* des modèles de conception.

Pensez-y, quelque chose a changé depuis la dernière fois que nous avons parlé de vocabulaires partagés ; vous avez maintenant commencé à construire votre propre vocabulaire ! Sans oublier que vous avez également appris un ensemble complet de principes de conception OO à partir desquels vous pouvez facilement comprendre la motivation et le fonctionnement de tout nouveau modèle que vous rencontrez.

Maintenant que vous maîtrisez les bases du modèle de conception, il est temps de diffuser l'information aux autres. Pourquoi ? Parce que lorsque vos collègues développeurs connaissent les modèles et utilisent également un vocabulaire commun, cela conduit à de meilleures conceptions et à une meilleure communication, et, mieux encore, cela vous fera gagner beaucoup de temps que vous pourrez consacrer à des choses plus intéressantes.



Les cinq meilleures façons de partager votre vocabulaire

- 1. Lors des réunions de conception :** lorsque vous rencontrez votre équipe pour discuter de la conception d'un logiciel, utilisez des modèles de conception pour vous aider à rester « dans le coup » plus longtemps. En discutant des conceptions du point de vue des modèles de conception et des principes OO, vous évitez à votre équipe de s'enliser dans les détails de mise en œuvre et évitez de nombreux malentendus.
- 2. Avec d'autres développeurs :** Utilisez des modèles dans vos discussions avec d'autres développeurs. Cela permet aux autres développeurs d'en apprendre davantage sur de nouveaux modèles et de créer une communauté. Le meilleur dans le partage de ce que vous avez appris, c'est ce sentiment formidable que vous éprouvez lorsque quelqu'un d'autre « comprend » !
- 3. Dans la documentation d'architecture :** lorsque vous rédigez une documentation architecturale, l'utilisation de modèles réduira la quantité de documentation que vous devez rédiger et donnera au lecteur une image plus claire de la conception.
- 4. Dans les commentaires de code et les conventions de dénomination :** lorsque vous écrivez du code, identifiez clairement les modèles que vous utilisez dans les commentaires. Choisissez également des noms de classe et de méthode qui révèlent les modèles sous-jacents. Les autres développeurs qui doivent lire votre code vous remercieront de leur permettre de comprendre rapidement votre implémentation.
- 5. Aux groupes de développeurs intéressés :** Partagez vos connaissances. De nombreux développeurs ont entendu parler des modèles, mais n'ont pas une bonne compréhension de ce qu'ils sont. Portez-vous volontaire pour offrir un déjeuner sur les modèles ou une conférence dans votre groupe d'utilisateurs local.



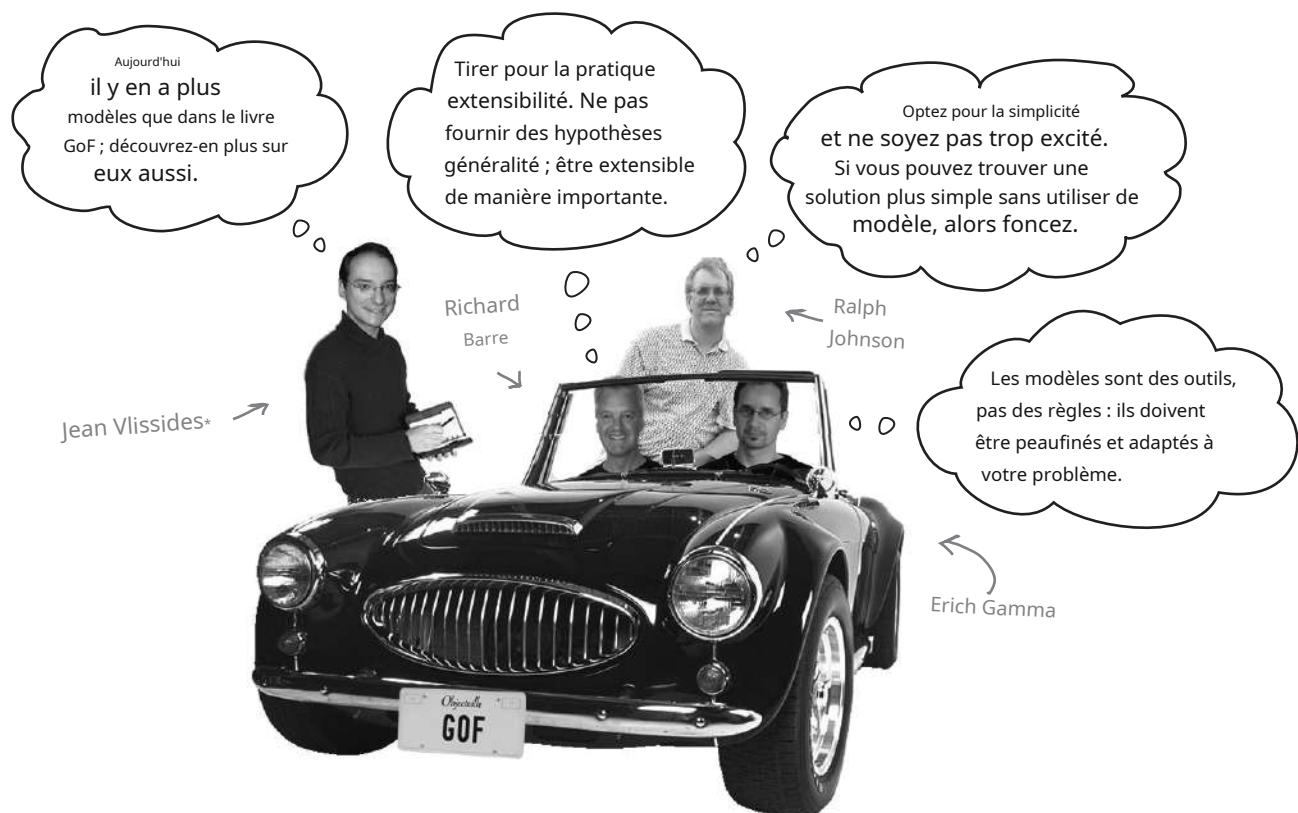
Croisière à Objectville avec la bande des quatre

Vous ne trouverez pas les Jets ou les Sharks dans les parages d'Objectville, mais vous trouverez le Gang des Quatre. Comme vous l'avez probablement remarqué, vous ne pouvez pas aller bien loin dans le Monde des Motifs sans les croiser. Alors, qui est ce mystérieux gang ?

En termes simples, « le GoF », qui comprend Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, est le groupe de gars qui a mis au point le premier catalogue de modèles et, ce faisant, a lancé tout un mouvement dans le domaine du logiciel !

Comment ont-ils obtenu ce nom ? Personne ne le sait avec certitude ; c'est juste un nom qui est resté. Mais réfléchissez-y : si vous devez avoir un « élément de gang » qui court à Objectville, pourriez-vous penser à un groupe de gars plus sympa ? En fait, ils ont même accepté de nous rendre visite...

Le GoF a lancé le mouvement des modèles logiciels, mais de nombreuses autres personnes ont apporté des contributions significatives, notamment Ward Cunningham, Kent Beck, Jim Coplien, Grady Booch, Bruce Anderson, Richard Gabriel, Doug Lea, Peter Coad et Doug Schmidt, pour n'en citer que quelques-uns.



* John Vlissides est décédé en 2005. Une grande perte pour la communauté des Design Patterns.

Votre voyage ne fait que commencer...

Maintenant que vous maîtrisez les modèles de conception et que vous êtes prêt à approfondir vos connaissances, nous avons trois textes définitifs que vous devez ajouter à votre bibliothèque...



Le texte définitif sur les modèles de conception

C'est le livre qui a lancé tout le domaine des Design Patterns lors de sa sortie en 1995. Vous y trouverez tous les modèles fondamentaux. En fait, ce livre est la base de l'ensemble des modèles que nous avons utilisés dans *Modèles de conception de la tête la première*.

Vous ne constaterez pas que ce livre constitue le dernier mot sur les modèles de conception (le domaine s'est considérablement développé depuis sa publication), mais il est le premier et le plus définitif.

Je récupère un exemplaire de *Modèles de conception* est un excellent moyen de commencer à explorer les modèles après Head First.

Les auteurs de Design Patterns sont affectueusement surnommés le « Gang of Four », ou GoF en abrégé.

Christopher Alexander a inventé des modèles, qui ont inspiré l'application de solutions similaires aux logiciels.

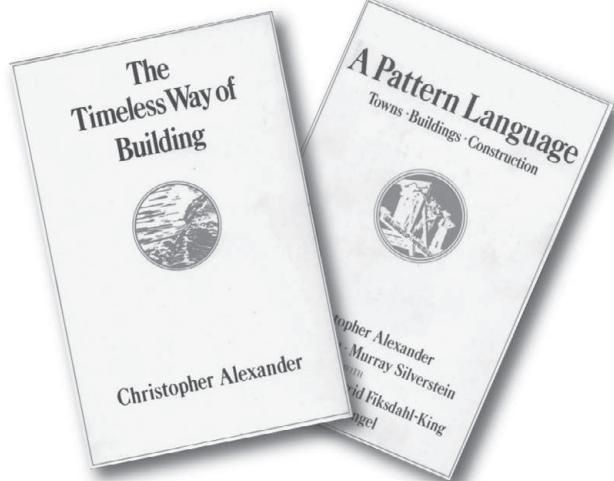


Les textes définitifs de Patterns

Les modèles n'ont pas commencé avec le GoF ; ils ont commencé avec Christopher Alexander, professeur d'architecture à Berkeley - c'est vrai, Alexander est un architecte, pas un informaticien. Alexander a inventé des modèles pour construire des architectures vivantes (comme des maisons, des villes et des cités).

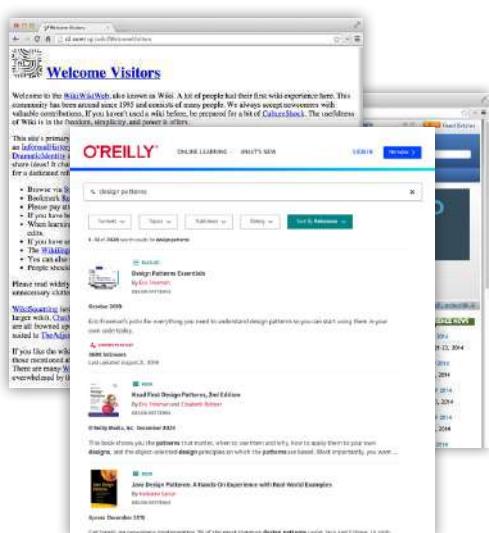
La prochaine fois que vous avez envie d'une lecture profonde et captivante, prenez *La manière intemporelle de construire* et *Un langage de modèles* vous verrez les véritables débuts des modèles de conception et reconnaîtrez les analogies directes entre la création d'une « architecture vivante » et d'un logiciel flexible et extensible.

Alors prenez une tasse de café Starbuzz, asseyez-vous et profitez...



Autres ressources sur les modèles de conception

Vous allez découvrir qu'il existe une communauté dynamique et amicale d'utilisateurs et d'auteurs de modèles et ils seront heureux de vous compter parmi eux. Voici quelques ressources pour vous aider à démarrer...



Sites Web

Le référentiel des modèles de Portland, dirigé par Ward Cunningham, est un wiki consacré à tout ce qui concerne les modèles. Vous y trouverez des fils de discussion sur tous les sujets auxquels vous pouvez penser en rapport avec les modèles et les systèmes OO.

c2.com/cgi/wiki?BienvenueVisiteurs

Le Groupe Hillside favorise les pratiques de programmation et de conception communes et fournit une ressource centrale pour le travail sur les modèles. Le site comprend des informations sur de nombreuses ressources liées aux modèles, telles que des articles, des livres, des listes de diffusion et des outils.

hillside.net

Apprentissage en ligne O'Reilly propose des livres, des cours et des cours en direct sur les modèles de conception en ligne. Vous trouverez également un cours intensif sur les modèles de conception basé sur ce livre.

oreilly.com



Conférences et ateliers

Si vous souhaitez interagir avec la communauté des modèles, n'hésitez pas à consulter les nombreuses conférences et ateliers liés aux modèles. Le site Hillside en tient une liste complète. Consultez Pattern Languages of Programs (PLoP) et la conférence ACM sur les systèmes, langages et applications orientés objet (OOPSLA), qui fait désormais partie de la conférence SPLASH.

Autres ressources

Nous aurions tort de ne pas mentionner Google, Stack Overflow, Quora et de nombreux autres sites et services comme de bons endroits pour poser des questions, trouver des réponses et discuter de modèles de conception. Comme pour tout ce qui se trouve sur le Web, vérifiez toujours les informations que vous recevez.

Le zoo des motifs

Comme vous venez de le voir, les motifs n'ont pas commencé avec les logiciels, mais avec l'architecture des bâtiments et des villes. En fait, le concept de motifs peut être appliqué dans de nombreux domaines différents. Promenez-vous dans le Zoo des motifs pour en voir quelques-uns...



Modèles architecturaux sont utilisés pour créer l'architecture vivante et dynamique des bâtiments, des villes et des cités. C'est là que les motifs ont vu le jour.



Habitat : se trouve dans les bâtiments dans lesquels vous aimez vivre, regarder et visiter.

Habitat : vu traîner autour des architectures à trois niveaux, des systèmes client-serveur et du Web.



Modèles d'applications sont modèles pour créer architecture au niveau du système.

Plusieurs multi-niveaux les architectures tombent dans cette catégorie

catégorie.



Note de terrain : MVC est connu pour passer pour un modèle d'application.



Modèles spécifiques au domaine sont des modèles qui concernent des problèmes dans des domaines spécifiques, comme les systèmes concurrents ou les systèmes en temps réel.

Aidez-nous à trouver un habitat

Informatique d'entreprise

Modèles de processus d'entreprise
décrire l'interaction entre les entreprises, les clients, et des données, et peut être appliquée à des problèmes tels que la manière de fabriquer et de communiquer les décisions.



On les voit traîner dans les salles de conseil d'entreprise et dans les projets réunions de direction.

Aidez-nous à trouver un habitat

Équipe de développement

Équipe de support client



Modèles d'organisation
décrire les structures et les pratiques de l'homme organisations. La plupart les efforts déployés jusqu'à présent se sont concentrés sur les organisations qui produisent et/ou logiciel de support.



Interface utilisateur

Modèles de conception

s'adresser à la problèmes de comment faire conception interactive programmes logiciels.



Habitat : observé à proximité des concepteurs de jeux vidéo, des créateurs d'interfaces graphiques et des producteurs.

Notes de terrain : veuillez ajouter ici vos observations de domaines de modèles :

Annihiler le mal avec des anti-modèles

L'Univers ne serait tout simplement pas complet si nous avions des modèles et pas d'anti-modèles, n'est-ce pas ?

Si un modèle de conception vous donne une solution générale à un problème récurrent dans un contexte particulier, alors que vous apportez un anti-modèle ?

Un **Anti-modèle** vous explique comment passer d'un problème à une MAUVAISE solution.

Vous vous demandez probablement : « Pourquoi diable quelqu'un perdrait-il son temps à documenter de mauvaises solutions ? »

Pensez-y de cette façon : s'il existe une mauvaise solution récurrente à un problème courant, alors en la documentant, nous pouvons empêcher d'autres développeurs de faire la même erreur. Après tout, éviter les mauvaises solutions peut être tout aussi précieux que d'en trouver de bonnes !

Regardons les éléments d'un anti-modèle :

Un anti-modèle vous indique pourquoi une mauvaise solution est attrayante. Soyons honnêtes, personne ne choisirait une mauvaise solution si elle n'avait pas quelque chose d'attrayant au premier abord. L'une des principales fonctions de l'anti-modèle est de vous alerter sur l'aspect séduisant de la solution.

Un anti-modèle vous indique pourquoi cette solution est mauvaise à long terme. Pour comprendre pourquoi il s'agit d'un anti-modèle, vous devez comprendre comment il va avoir un effet négatif à long terme. L'anti-modèle décrit les situations dans lesquelles vous rencontrerez des problèmes en utilisant la solution.

Un anti-modèle suggère d'autres modèles applicables qui peuvent fournir de bonnes solutions. Pour être vraiment utile, un anti-modèle doit vous orienter dans la bonne direction ; il doit suggérer d'autres possibilités qui peuvent conduire à de bonnes solutions.

Jetons un œil à un anti-modèle.



Un anti-modèle toujours

Cela semble être une bonne solution, mais s'avère être une mauvaise solution lorsqu'elle est appliquée.

En documentant les anti-modèles que nous aidons d'autres à reconnaître les mauvaises solutions avant qu'elles ne les mettre en œuvre.

Comme les motifs, il y a il existe de nombreux types des anti-modèles y compris le développement, OO, organisationnel, et spécifique au domaine anti-modèles.

Voici un exemple d'anti-modèle de développement logiciel.



Anti-modèle

Nom: Marteau d'or

Problème: Vous devez choisir des technologies pour votre développement et vous pensez qu'une seule technologie doit dominer l'architecture.

Contexte: Vous devez développer un nouveau système ou un nouveau logiciel qui ne s'adapte pas bien à la technologie que connaît l'équipe de développement.

Forces :

- L'équipe de développement s'engage envers la technologie qu'elle connaît.
- L'équipe de développement n'est pas familière avec d'autres technologies.
- Les technologies inconnues sont considérées comme risquées.
- Il est facile de planifier et d'estimer le développement en utilisant la technologie familiale.

Solution supposée : Utilisez quand même la technologie que vous connaissez. La technologie est appliquée de manière obsessionnelle à de nombreux problèmes, y compris dans des domaines où elle est clairement inappropriée.

Solution refactorisée : Développez les connaissances des développeurs grâce à l'éducation, à la formation et à des groupes d'étude de livres qui exposent les développeurs à de nouvelles solutions.

Exemples :

Les entreprises Web continuent d'utiliser et de maintenir leurs systèmes de mise en cache internes lorsque des alternatives open source sont utilisées.

Tout comme un modèle de conception, un anti-modèle a un nom afin que nous puissions créer un vocabulaire partagé.

Le problème et le contexte, tout comme une description de modèle de conception.

Vous dit pourquoi la solution est attractif.

La mauvaise solution, mais néanmoins attrayante.

Comment se rendre à un bonne solution.

Exemple où cet anti-modèle a été observé.

Adapté du wiki du Portland Pattern Repository sur <https://wiki.c2.com/?WelcomeVisitors> où vous trouverez de nombreux anti-modèles et discussions.



Des outils pour votre boîte à outils de conception

Vous avez atteint ce point où vous nous avez dépassés. Il est maintenant temps de sortir dans le monde et d'explorer les modèles par vous-même...

Notions de base sur OO

- Abstraction
- Encapsulation
- Polymorphisme
- Héritage
- Le temps
- vous de
- découvrir
- par vous-
- nombreux
- des modé
- pas ment
- des modé
- ceux que
- abordés.
- vos prop

Principes OO

- Encapsuler ce qui varie.
- Privilégiez la composition plutôt que l'héritage.
- Programmez vers des interfaces, pas vers des implémentations.
- Efforcez-vous d'obtenir des conceptions faiblement couplées entre les objets qui interagissent.
- Les cours devraient être ouverts pour prolongation mais fermés pour modification.
- Dépendez des abstractions. Ne dépendez pas de classes concrètes.
- Parlez uniquement à vos amis.
- Ne nous appelez pas, nous vous appellerons.
- Une classe ne devrait avoir qu'une seule raison de changer.

The image shows a stack of cards with various text snippets related to Object-Oriented Programming (OOP) models:

- A large card at the top left reads "Modèles OO" and "StO".
- To its right, another card says "Vos modèles ici !".
- Below these, a card features the text "Motifs composés".
- Further down, a card contains the sentence "résout un problème récurrent_ou_l_g_et_ne_l_au_p_l_d'accord_le_m_".
- Other cards in the stack have text that is mostly illegible due to rotation or overlap, but includes snippets like "I'dunA", "encdarpegeht", "intwedihr", "varydaelisn", "afuuuutno", "sssuuuupppp", and "lqqaqtuunsuiesace".



BULLET POINTS

- f** Laissez les modèles de conception émerger dans vos créations ; ne les forcez pas simplement pour le plaisir d'utiliser un modèle.
 - f** Les modèles de conception ne sont pas gravés dans la pierre ; adaptez-les et modifiez-les pour répondre à vos besoins.
 - f** Utilisez toujours la solution la plus simple qui répond à vos besoins, même si elle n'inclut pas de modèle.
 - f** Modèles de conception d'études des catalogues pour vous familiariser avec les modèles et les relations entre eux.
 - f** Les classifications de modèles (ou catégories) fournissent des regroupements de modèles. Lorsqu'elles vous aident, utilisez-les.
 - f** Vous devez vous engager à devenir rédacteur de modèles : cela prend du temps et de la patience, et vous devrez être prêt à faire beaucoup de peaufinage.
 - f** Vous ne serez pas le seul à utiliser les modèles existants, et vous rencontrerez seront des adaptations de modèles existants, et pourra de nouveaux modèles.
 - f** Développez le vocabulaire commun de votre équipe. C'est l'un des avantages les plus importants de l'utilisation de modèles.
 - f** Comme toute communauté, la communauté des patrons a son propre jargon. Ne vous laissez pas décourager. Après avoir lu ce livre, vous en connaîtrez désormais la plupart.

En quittant Objectville...



Garçon, ça a été génial de t'avoir à Objectville.

Tu vas nous manquer, c'est sûr. Mais ne t'inquiète pas, avant même de t'en rendre compte, le prochain livre de Head First sortira et tu pourras nous rendre visite à nouveau. Quel est le prochain livre, me demandes-tu ? Hmm, bonne question ! Pourquoi ne nous aides-tu pas à décider ? Envoyez un e-mail à booksuggestions@wickedlysmart.com .



Associez chaque motif à sa description :

Modèle	Description
Décorateur	Enveloppe un objet et lui fournit une interface différente.
État	Les sous-classes décident comment implémenter les étapes d'un algorithme.
Itérateur	Les sous-classes décident quelles classes concrètes créer.
Façade	Garantit qu'un seul et unique objet est créé.
Stratégie	Encapsule les comportements interchangeables et utilise la délégation pour décider lequel utiliser.
Procuration	Les clients traitent les collections d'objets et les objets individuels de manière uniforme.
Méthode d'usine	Encapsule les comportements basés sur l'état et utilise la délégation pour basculer entre les comportements.
Adaptateur	Fournit un moyen de parcourir une collection d'objets sans exposer son implémentation.
Observateur	Simplifie l'interface d'un ensemble de classes. Encapsule un objet pour fournir un nouveau comportement.
Méthode de modèle	Permet à un client de créer des familles d'objets sans spécifier leurs classes concrètes.
Composite	Permet aux objets d'être notifiés lorsque leur état change.
Singleton	Encapsule un objet pour contrôler l'accès à celui-ci.
Usine abstraite	Encapsule une requête sous forme d'objet.
Commande	

Modèles restants



Tout le monde ne peut pas être le plus populaire. Beaucoup de choses ont changé au cours des 25 dernières années. Depuis *Modèles de conception : éléments d'un logiciel orienté objet réutilisable* Depuis leur première sortie, les développeurs ont appliqué ces modèles des milliers de fois. Les modèles que nous résumons dans cette annexe sont des modèles GoF officiels à part entière, porteurs de cartes, mais ne sont pas utilisés aussi souvent que les modèles que nous avons explorés jusqu'à présent. Mais ces modèles sont géniaux en eux-mêmes, et si votre situation l'exige, vous devriez les appliquer la tête haute. Notre objectif dans cette annexe est de vous donner une idée générale de ce que sont ces modèles.

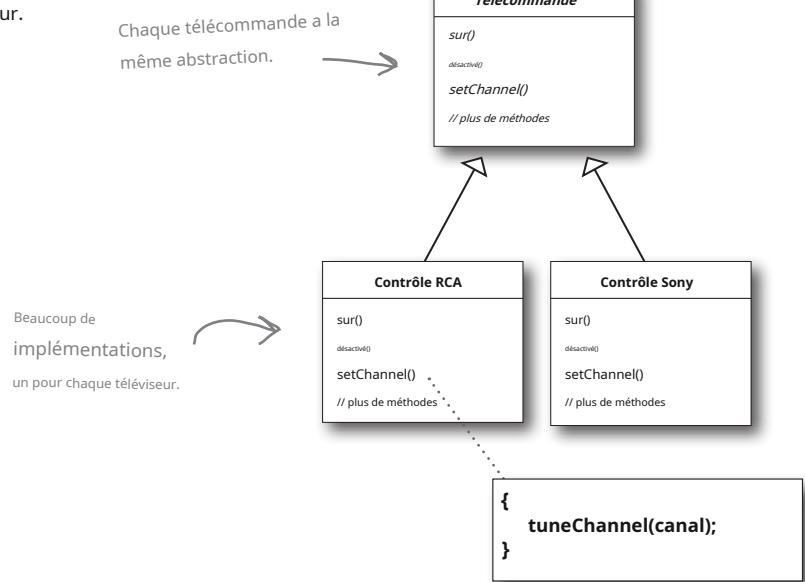
Pont

Utilisez le modèle Bridge pour varier non seulement vos implémentations, mais également vos abstractions.

Un scénario

Imaginez que vous écrivez le code d'une nouvelle télécommande ergonomique et conviviale pour téléviseurs. Vous savez déjà que vous devez utiliser de bonnes techniques orientées objet, car même si la télécommande est basée sur le même *abstraction*, il y aura beaucoup de *implémentations*—un pour chaque modèle de téléviseur.

Il s'agit d'une abstraction. Il peut s'agir d'une interface ou d'une classe abstraite.



Votre dilemme

Vous savez que l'interface utilisateur de la télécommande ne sera pas parfaite du premier coup. En fait, vous vous attendez à ce que le produit soit peaufiné à plusieurs reprises au fur et à mesure que des données d'utilisabilité sont collectées sur la télécommande.

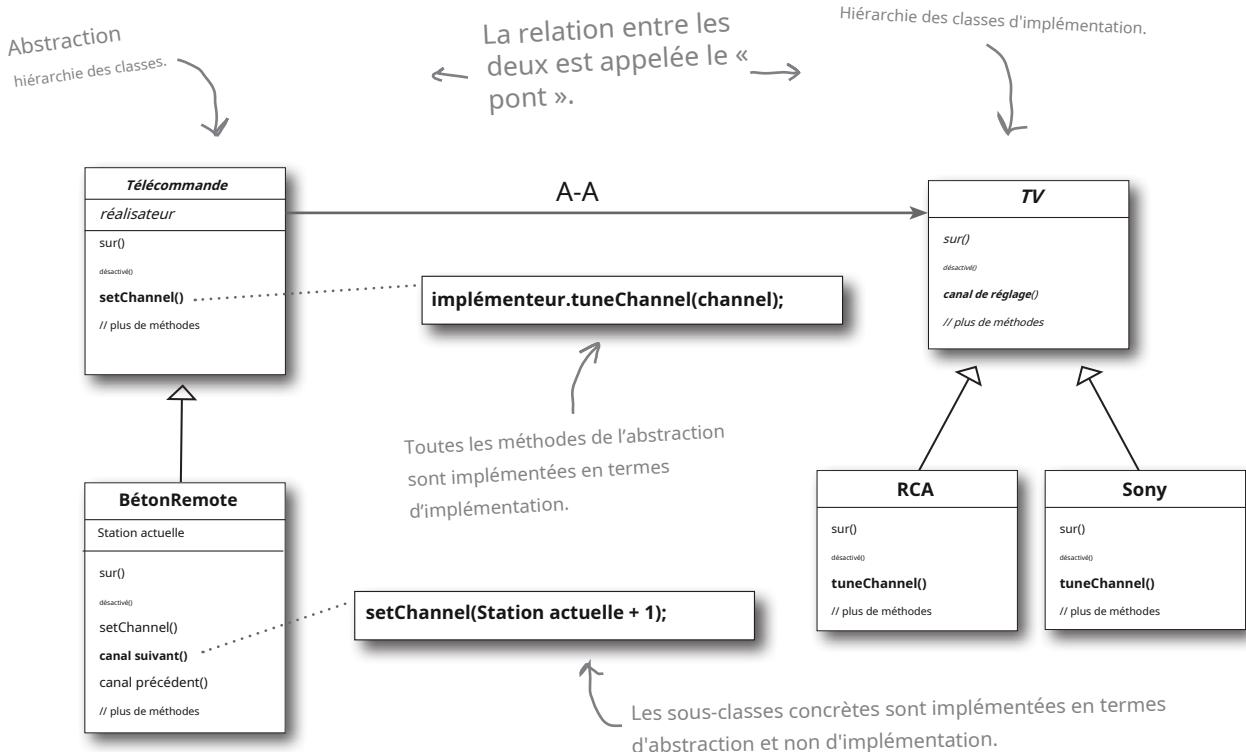
Donc votre dilemme est que les télécommandes vont changer et que les téléviseurs vont changer. Vous avez déjà *distrait* l'interface utilisateur pour que vous puissiez faire varier les *mises en œuvre* sur les nombreux téléviseurs que vos clients posséderont. Mais vous devrez également *varier l'abstraction* car cela va changer au fil du temps à mesure que la télécommande est améliorée en fonction des commentaires des utilisateurs.

Alors, comment allez-vous créer une conception orientée objet qui vous permet de varier l'implémentation *et* l'abstraction?

Grâce à cette conception, nous pouvons uniquement modifier l'implémentation du téléviseur, et non l'interface utilisateur.

Pourquoi utiliser le modèle Bridge ?

Le modèle Bridge vous permet de varier l'implémentation et l'abstraction en plaçant les deux dans des hiérarchies de classes distinctes.



Vous disposez désormais de deux hiérarchies, une pour les télécommandes et une autre pour les implémentations TV spécifiques à la plate-forme. Le pont vous permet de faire varier indépendamment chaque côté des deux hiérarchies.

Avantages du pont

- ❖ Découpe une implémentation afin qu'elle ne soit pas liée de manière permanente à une interface.
- ❖ L'abstraction et la mise en œuvre peuvent être étendues indépendamment.
- ❖ Les modifications apportées aux classes d'abstraction concrètes n'affectent pas le client.

Utilisations et inconvénients des ponts

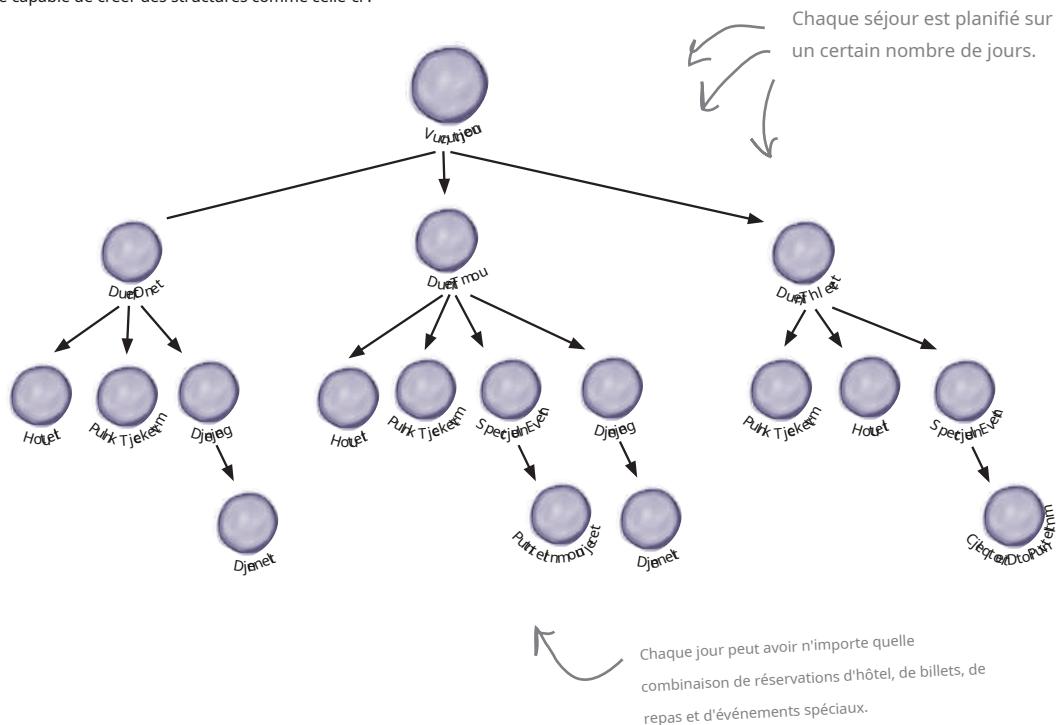
- ❖ Utile dans les systèmes graphiques et de fenêtrage qui doivent fonctionner sur plusieurs plates-formes.
- ❖ Utile à chaque fois que vous avez besoin de faire varier une interface et une implémentation de différentes manières.
- ❖ Augmente la complexité.

Constructeur

Utilisez le modèle Builder pour encapsuler la construction d'un produit et lui permettre d'être construit par étapes.

Un scénario

On vient de vous demander de créer un planificateur de vacances pour Patternsland, un nouveau parc à thème situé juste à l'extérieur d'Objectville. Les visiteurs du parc peuvent choisir un hôtel et différents types de billets d'entrée, faire des réservations de restaurant et même réserver des événements spéciaux. Pour créer un planificateur de vacances, vous devez être capable de créer des structures comme celle-ci :



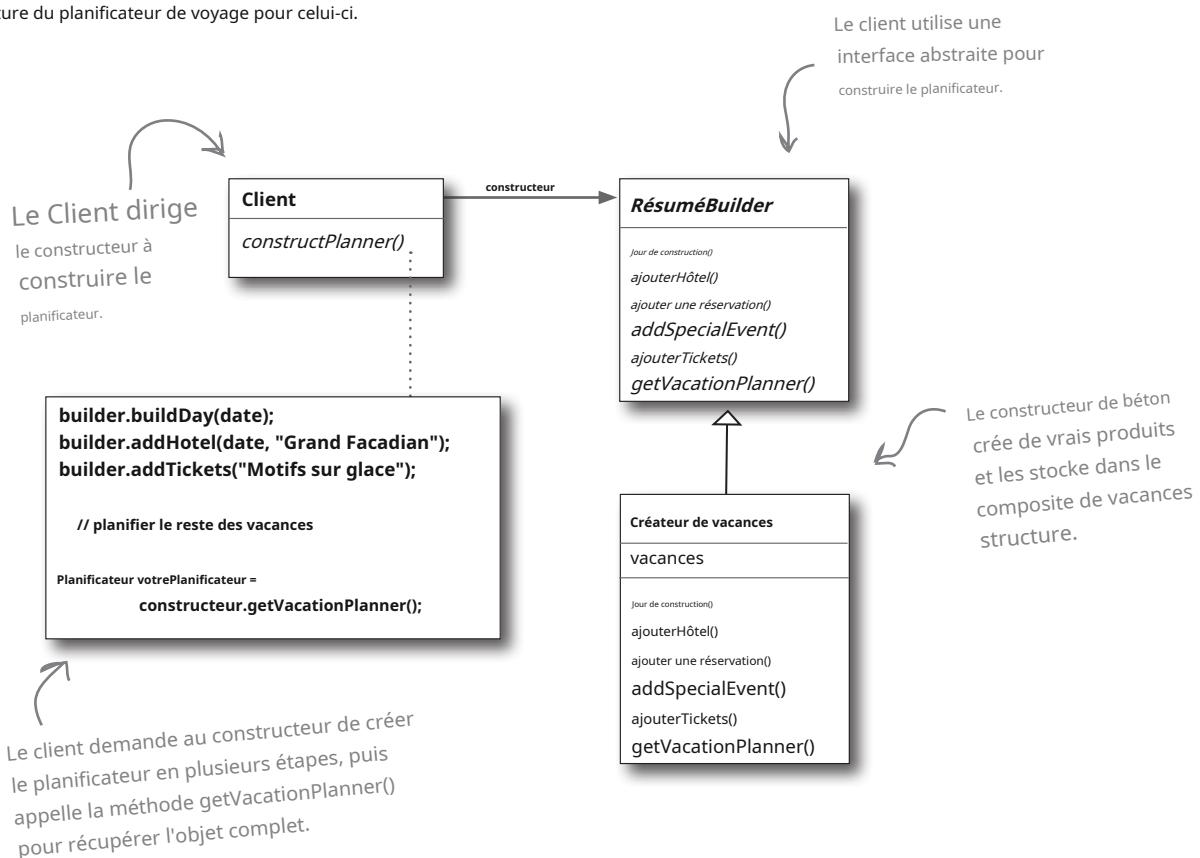
Vous avez besoin d'une conception flexible

Le planning de chaque invité peut varier en fonction du nombre de jours et du type d'activités qu'il comprend. Par exemple, un résident local peut ne pas avoir besoin d'un hôtel, mais souhaite faire des réservations pour le dîner et les événements spéciaux. Un autre invité peut arriver par avion à Objectville et avoir besoin d'un hôtel, de réservations pour le dîner et de billets d'entrée.

Vous avez donc besoin d'une structure de données flexible capable de représenter les planificateurs d'invités et toutes leurs variantes. Vous devez également suivre une séquence d'étapes potentiellement complexes pour créer le planificateur. Comment pouvez-vous fournir un moyen de créer la structure complexe sans la mélanger avec les étapes de sa création ?

Pourquoi utiliser le modèle Builder ?

Vous vous souvenez d'Iterator ? Nous avons encapsulé l'itération dans un objet séparé et masqué la représentation interne de la collection au client. C'est la même idée ici : nous encapsulons la création du planificateur de voyage dans un objet (appelons-le un générateur) et demandons à notre client de demander au générateur de construire la structure du planificateur de voyage pour celui-ci.



Avantages pour les constructeurs

- ❖ Encapsule la manière dont un objet complexe est construit.
- ❖ Permet de construire des objets dans un processus en plusieurs étapes et variable (par opposition aux usines en une seule étape).
- ❖ Masque la représentation interne du produit au client.
- ❖ Les implémentations de produits peuvent être échangées car le client ne voit qu'une interface abstraite.

Utilisations et inconvénients des constructeurs

- ❖ Souvent utilisé pour la construction de structures composites.
- ❖ La construction d'objets nécessite une connaissance du domaine client plus poussée que lors de l'utilisation d'une Factory.

Chaîne de responsabilité

Utilisez le modèle de chaîne de responsabilité lorsque vous souhaitez donner à plusieurs objets la possibilité de gérer une demande.

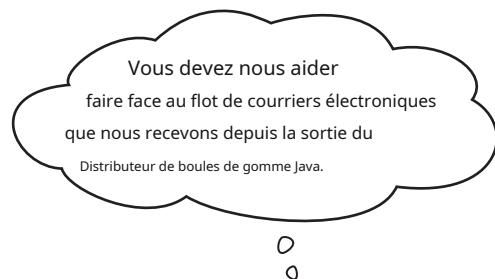
Un scénario

Depuis la sortie de la machine Gumball fonctionnant sous Java, Mighty Gumball reçoit plus de courrières électroniques qu'il ne peut en gérer. Selon leur propre analyse, ils reçoivent quatre types de courrières électroniques : des courrières de fans de clients qui adorent le nouveau jeu 1-in-10, des plaintes de parents dont les enfants sont accros au jeu, des demandes d'installation de machines à de nouveaux endroits et une bonne quantité de spam.

Tous les courrières des fans doivent être adressés directement au PDG, toutes les plaintes doivent être adressées au service juridique et toutes les demandes de nouvelles machines doivent être adressées au service de développement commercial. Les spams doivent être supprimés.

Votre tâche

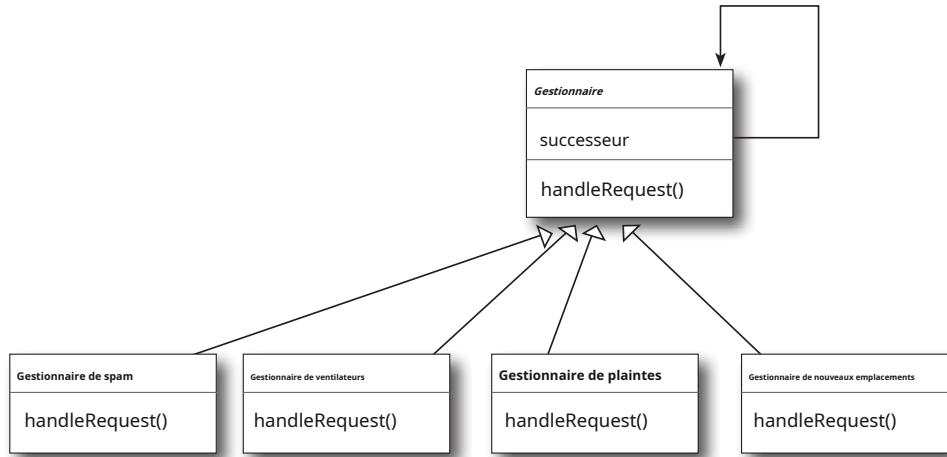
Mighty Gumball a déjà écrit des détecteurs d'IA qui peuvent dire si un e-mail est un spam, un courrier de fan, une plainte ou une demande, mais ils ont besoin que vous créez une conception qui puisse utiliser les détecteurs pour gérer les e-mails entrants.



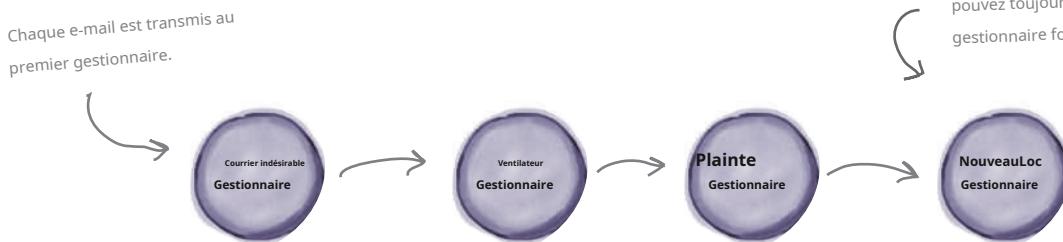
Comment utiliser le modèle de chaîne de responsabilité

Avec le modèle de chaîne de responsabilité, vous créez une chaîne d'objets pour examiner les demandes. Chaque objet examine à son tour une demande et la traite ou la transmet à l'objet suivant de la chaîne.

Chaque objet de la chaîne agit comme un gestionnaire et possède un objet successeur. S'il peut gérer la demande, il le fait ; sinon, il transmet la demande à son successeur.



Lorsqu'un e-mail est reçu, il est transmis au premier gestionnaire : SpamHandler. Si le SpamHandler ne peut pas gérer la requête, elle est transmise au FanHandler. Et ainsi de suite...



Le courrier électronique n'est pas traité s'il tombe à la fin de la chaîne, même si vous pouvez toujours implémenter un gestionnaire fourre-tout.

Avantages de la chaîne de responsabilité

- ❖ Découpe l'expéditeur de la requête et ses destinataires.
- ❖ Simplifie votre objet car il n'a pas besoin de connaître la structure de la chaîne et de conserver des références directes à ses membres.
- ❖ Permet d'ajouter ou de supprimer des responsabilités de manière dynamique en modifiant les membres ou l'ordre de la chaîne.

Utilisations et inconvénients de la chaîne de responsabilité

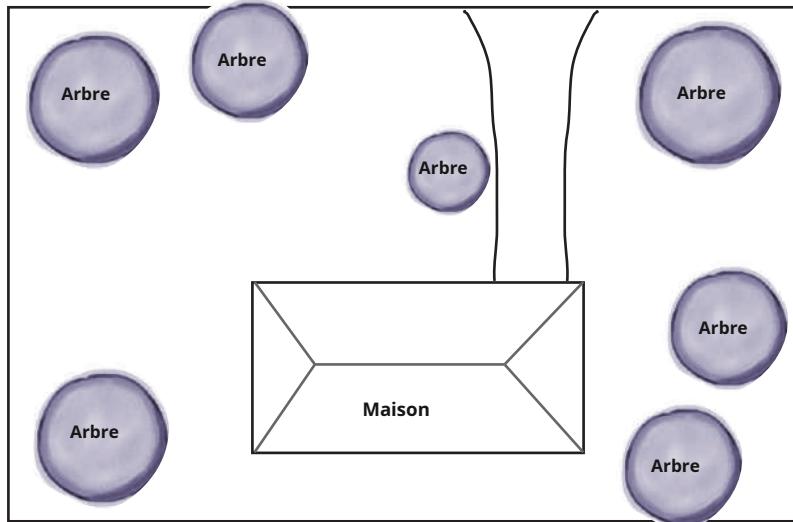
- ❖ Couramment utilisé dans les systèmes Windows pour gérer des événements tels que les clics de souris et les événements de clavier.
- ❖ L'exécution de la requête n'est pas garantie ; elle peut tomber en fin de chaîne si aucun objet ne la gère (cela peut être un avantage ou un inconvénient).
- ❖ Peut être difficile à observer et à déboguer lors de l'exécution.

Poids mouche

Utilisez le modèle Flyweight lorsqu'une instance d'une classe peut être utilisée pour fournir plusieurs instances virtuelles.

Un scénario

Vous souhaitez ajouter des arbres en tant qu'objets dans votre nouvelle application de conception de paysage. Dans votre application, les arbres ne font pas grand-chose ; ils ont un emplacement XY et peuvent se dessiner de manière dynamique, en fonction de leur âge. Le fait est qu'un utilisateur peut vouloir avoir beaucoup, beaucoup d'arbres dans l'une de ses conceptions de paysage domestique. Cela pourrait ressembler à ceci :



Chaque instance d'arbre maintient son propre état.

Arbre
xCoord
yCoord
âge
afficher() {
// utiliser les coordonnées
XY // et l'âge complexe //
calculs associés
}

Le dilemme de votre gros client

Vous avez un client clé avec qui vous discutez depuis des mois. Il va acheter 1 000 licences de votre application et utiliser votre logiciel pour concevoir le paysage de vastes communautés planifiées. Après avoir utilisé votre logiciel pendant une semaine, votre client se plaint que lorsqu'il crée de grands bosquets d'arbres, l'application commence à devenir lente...

Pourquoi utiliser le modèle Flyweight ?

Et si, au lieu d'avoir des milliers d'objets Tree, vous pouviez repenser votre système de manière à n'avoir qu'une seule instance de Tree et un objet client qui gère l'état de TOUS vos arbres ? C'est le poids mouche !

Tout l'état, pour TOUS vos objets d'arbre virtuel, est stocké dans ce tableau 2D.



```
Gestionnaire d'arbres
arbreArray

afficherArbres() {
    // pour tous les arbres {
        // obtenir l'affichage de la
        // ligne du tableau (x, y, âge);
    }
}
```

Un seul objet arbre sans état.

```
Arbre
affichage(x, y, âge) {
    // utiliser les coordonnées
    XY // et l'âge complexe //
    calculs associés
}
```



Avantages du poids mouche

- ❖ Réduit le nombre d'instances d'objet lors de l'exécution, économisant ainsi de la mémoire.
- ❖ Centralise l'état de nombreux objets « virtuels » dans un seul emplacement.

Utilisations et inconvénients des poids mousches

- ❖ Le Flyweight est utilisé lorsqu'une classe possède de nombreuses instances et qu'elles peuvent toutes être contrôlées de manière identique.
- ❖ L'inconvénient du modèle Flyweight est qu'une fois que vous l'avez implémenté, les instances uniques et logiques de la classe ne pourront pas se comporter indépendamment des autres instances.

Interprète

Utilisez le modèle d'interpréteur pour créer un interprète pour une langue.

Un scénario

Vous vous souvenez du simulateur de canard ? Vous avez l'impression qu'il constituerait également un excellent outil pédagogique pour que les enfants apprennent à programmer. Grâce au simulateur, chaque enfant peut contrôler un canard avec un langage simple. Voici un exemple de langage :

droite;
pendant que (jour) vole ;
charlatan;
... et puis cancan.

Tournez le canard à droite.
Voler toute la journée...



L'interprète
Le modèle nécessite
quelques connaissances de
grammaires formelles.

Si vous n'avez jamais étudié les grammaires
formelles, allez-y et lisez le modèle ; vous en
comprendrez quand même l'essentiel.

Maintenant, en vous rappelant comment créer des grammaires à partir d'un de vos anciens cours d'introduction à la programmation, vous écrivez la grammaire :

expression ::= <commande> | <séquence> | <récitation>
séquence ::= <expression> ';' <expression>
commande ::= droite | coin-coin | voler
récitation ::= while '(' <variable> ')' <expression> **variable ::=**
[AZ,az]+

Un programme est une expression composée
de séquences de commandes et de répétitions
(instructions « while »).

Une séquence est un ensemble
d'expressions séparées
par des points-virgules.

Nous avons trois
commandes : droite,
cancaner et voler.

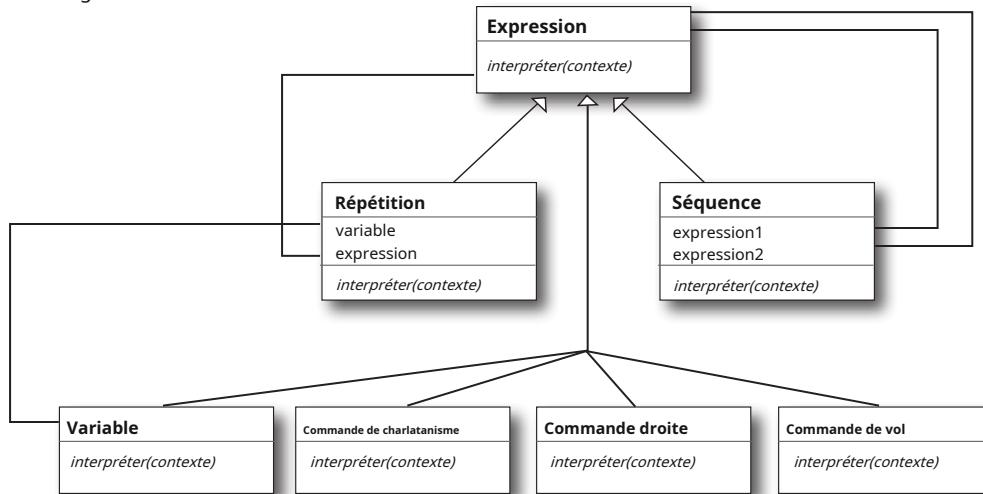
Une instruction while est
simplement une variable
conditionnelle et une expression.

Et maintenant ?

Vous avez une grammaire ; maintenant, tout ce dont vous avez besoin est un moyen de représenter et d'interpréter les phrases dans la grammaire afin que les élèves puissent voir les effets de leur programmation sur les canards simulés.

Comment mettre en œuvre un interprète

Lorsque vous devez implémenter un langage simple, le modèle Interpreter définit une représentation basée sur des classes pour sa grammaire ainsi qu'un interpréteur pour interpréter ses phrases. Pour représenter le langage, vous utilisez une classe pour représenter chaque règle du langage. Voici le langage duck traduit en classes. Notez le mappage direct vers la grammaire.



Pour interpréter le langage,appelez la méthode `interpret()` sur chaque type d'expression. Cette méthode reçoit un contexte (qui contient le flux d'entrée du programme que nous analysons) et fait correspondre l'entrée et l'évalue.

Avantages de l'interprète

- ❖ Représenter chaque règle de grammaire dans une classe rend le langage facile à implémenter.
- ❖ Parce que la grammaire est représentée par des classes, vous pouvez facilement modifier ou étendre le langage.
- ❖ En ajoutant des méthodes à la structure de la classe, vous pouvez ajouter de nouveaux comportements au-delà de l'interprétation, comme une jolie impression et une validation de programme plus sophistiquée.

Utilisations et inconvénients de l'interprète

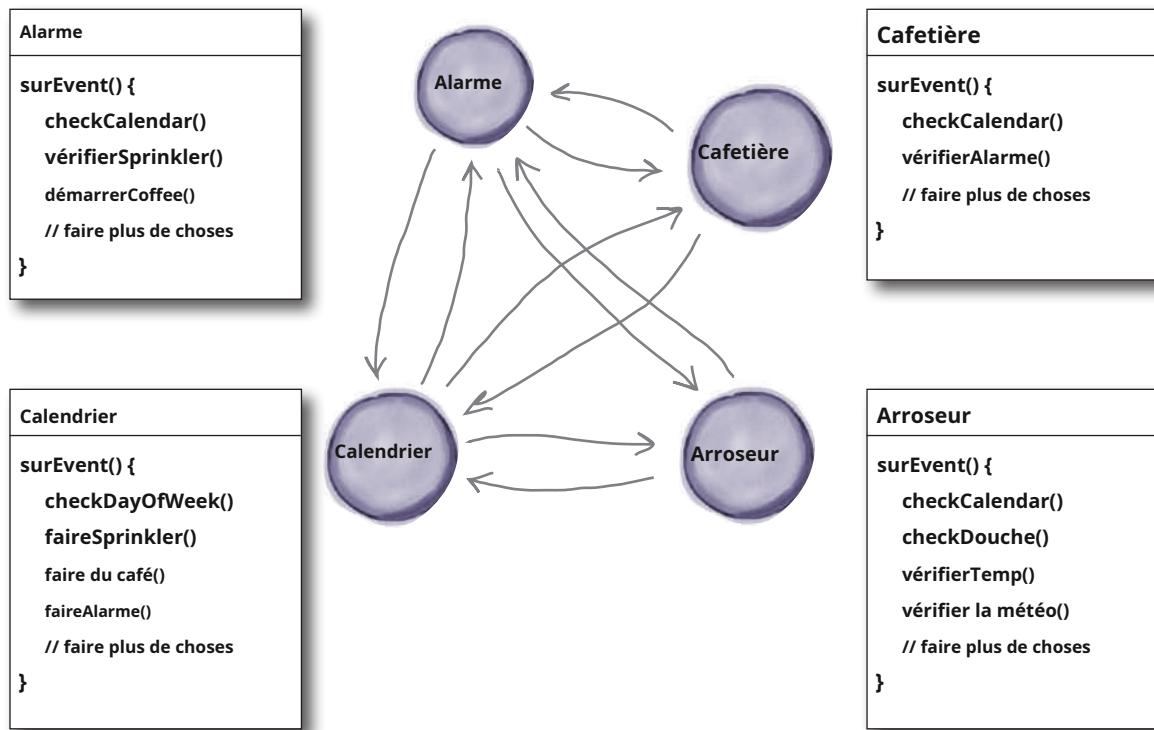
- ❖ Utilisez Interpreter lorsque vous devez implémenter un langage simple.
- ❖ Convient lorsque vous avez une grammaire simple et que la simplicité est plus importante que l'efficacité.
- ❖ Utilisé pour les langages de script et de programmation.
- ❖ Ce modèle peut devenir fastidieux lorsque le nombre de règles grammaticales est important. Dans ces cas, un générateur de type analyseur/compilateur peut être plus approprié.

Médiateur

Utilisez le modèle Mediator pour centraliser les communications complexes et le contrôle entre les objets associés.

Un scénario

Bob a une maison automatisée, grâce aux braves gens de HouseOfTheFuture. Tous ses appareils sont conçus pour lui faciliter la vie. Lorsque Bob arrête d'appuyer sur le bouton de répétition, son réveil indique à la cafetière de commencer à préparer le café. Même si la vie est belle pour Bob, lui et d'autres clients demandent toujours de nouvelles fonctionnalités : pas de café le week-end... éteindre l'arroseur 15 minutes avant l'heure prévue pour une douche... régler l'alarme tôt les jours où l'on jette les ordures...



Le dilemme de HouseOfTheFuture

Il devient vraiment difficile de suivre quelles règles résident dans quels objets et comment les différents objets doivent être liés les uns aux autres.

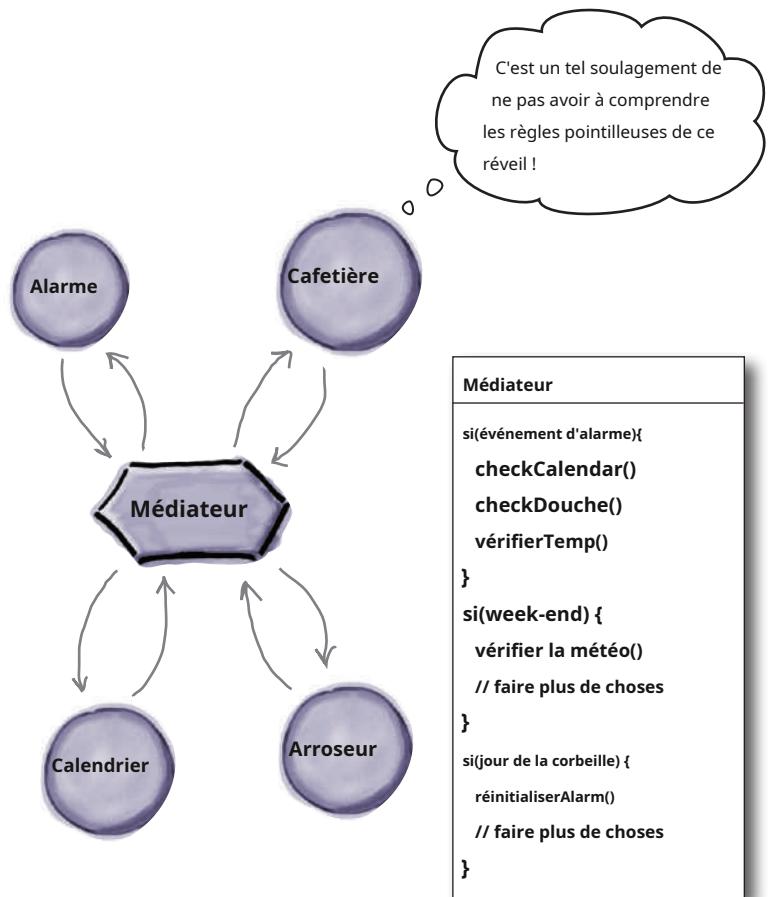
Médiateur en action...

Avec un médiateur ajouté au système, tous les objets de l'appareil peuvent être grandement simplifiés :

- ❖ Ils informent le Médiateur lorsque leur état changements.
- ❖ Ils répondent aux demandes des Médiateur.

Avant d'ajouter le médiateur, tous les objets de l'appareil devaient se connaître les uns les autres ; autrement dit, ils étaient tous étroitement couplés. Avec le médiateur en place, les objets de l'appareil sont tous complètement découplés les uns des autres.

Le médiateur contient toute la logique de contrôle de l'ensemble du système. Lorsqu'un appareil existant nécessite une nouvelle règle ou qu'un nouvel appareil est ajouté au système, vous savez que toute la logique nécessaire sera ajoutée au médiateur.



Avantages du médiateur

- ❖ Augmente la réutilisabilité des objets pris en charge par le Médiateur en les découpant du système.
- ❖ Simplifie la maintenance du système en centralisant la logique de contrôle.
- ❖ Simplifie et réduit la variété des messages envoyés entre les objets du système.

Utilisations et inconvénients du médiateur

- ❖ Le médiateur est couramment utilisé pour coordonner les composants d'interface utilisateur graphique associés.
- ❖ Un inconvénient du modèle Mediator est que sans une conception appropriée, l'objet Mediator lui-même peut devenir trop complexe.

Mémento

Utilisez le modèle Memento lorsque vous devez pouvoir ramener un objet à l'un de ses états précédents ; par exemple, si votre utilisateur demande une « annulation ».

Un scénario

Votre jeu de rôle interactif rencontre un énorme succès et a créé une légion d'addicts, tous essayant d'atteindre le légendaire « niveau 13 ». Au fur et à mesure que les utilisateurs progressent vers des niveaux de jeu plus difficiles, les chances de rencontrer une situation de fin de partie augmentent. Les fans qui ont passé des jours à progresser vers un niveau avancé sont naturellement vexés lorsque leur personnage est éliminé et qu'ils doivent tout recommencer. On réclame une commande « sauvegarder la progression », afin que les joueurs puissent stocker leur progression.

progression du jeu et au moins récupérer la plupart de leurs efforts lorsque leur personnage est injustement éteint. La fonction « sauvegarder la progression » doit être conçue pour renvoyer un joueur ressuscité au dernier niveau qu'il a terminé avec succès.



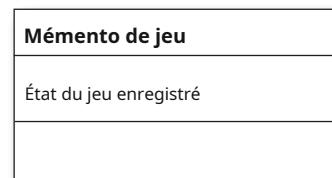
Le Memento au travail

Le Memento a deux objectifs :

- ❖ Sauvegarde de l'état important d'un objet clé d'un système

- ❖ Maintenir l'encapsulation de l'objet clé

En gardant à l'esprit le principe de responsabilité unique, il est également judicieux de conserver l'état que vous enregistrez séparément de l'objet clé. Cet objet distinct qui contient l'état est appelé objet Memento.



Client

```

// lorsqu'un nouveau niveau est atteint
Objet enregistré =
    (Objet) mgo.getCurrentState();

// lorsqu'une restauration est
requise mgo.restoreState(saved);

```

Bien que ce ne soit pas le cas
un terriblement fantaisiste
mise en œuvre, avis
que le Client n'a pas
accès aux données du
Memento.

Objet de jeu maître

```

Etat du jeu

Objet getCurrentState() {
    // rassembler l'état
    retour(état du jeu);
}

restoreState(Objet savedState) {
    // restaurer l'état
}

// faire d'autres trucs de jeu

```

Avantages de Memento

- ❖ Garder l'état enregistré à l'extérieur de l'objet clé permet de maintenir la cohésion.
- ❖ Conserve les données de l'objet clé encapsulées. Fournit une capacité de récupération facile à mettre en œuvre.

Utilisations et inconvénients des souvenirs

- ❖ Le Memento est utilisé pour sauvegarder l'état.
- ❖ L'inconvénient de l'utilisation de Memento est que la sauvegarde et la restauration de l'état peuvent prendre du temps.
- ❖ Dans les systèmes Java, envisagez d'utiliser la sérialisation pour enregistrer l'état d'un système.

Prototype

Utilisez le modèle de prototype lorsque la création d'une instance d'une classe donnée est coûteuse ou compliquée.

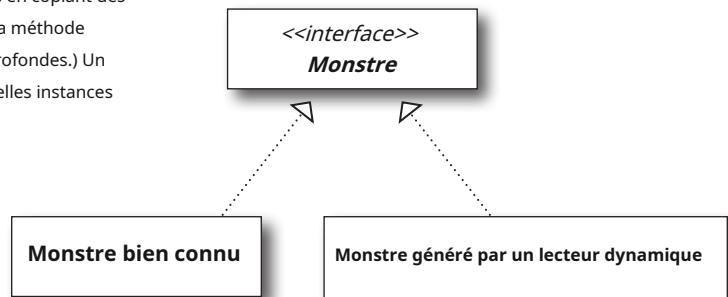
Un scénario

Votre jeu de rôle interactif a un appétit insatiable pour les monstres. Au cours de leur voyage à travers un paysage créé de manière dynamique, vos héros rencontrent une chaîne infinie d'ennemis qu'ils doivent maîtriser. Vous aimeriez que les caractéristiques des monstres évoluent avec le paysage changeant. Il n'est pas très logique que des monstres ressemblant à des oiseaux suivent vos personnages dans des royaumes sous-marins. Enfin, vous aimeriez permettre aux joueurs avancés de créer leurs propres monstres personnalisés.



Prototype à la rescousse

Le modèle Prototype vous permet de créer de nouvelles instances en copiant des instances existantes. (En Java, cela signifie généralement utiliser la méthode `clone()` ou la désérialisation lorsque vous avez besoin de copies profondes.) Un aspect clé de ce modèle est que le code client peut créer de nouvelles instances sans savoir quelle classe spécifique est en cours d'instanciation.



```

Créateur de monstres
=====
créer un monstre aléatoire () {
    Monstre m =
        MonsterRegistry.getMonster();
}
  
```

Le client a besoin d'un nouveau monstre adapté à la situation actuelle. (Le client ne saura pas quel genre de monstre il obtient.)

```

Registre des monstres
=====
Monstre getMonster() {
    // trouver le bon monstre return
    correctMonster.clone();
}
  
```

Le registre trouve le monstre approprié, en crée un clone et renvoie le clone.

Avantages du prototype

- ❖ Masque les complexités liées à la création de nouvelles instances à partir du client.
- ❖ Fournit la possibilité au client de générer des objets dont le type n'est pas connu.
- ❖ Dans certaines circonstances, la copie d'un objet peut être plus efficace que la création d'un nouvel objet.

Utilisations et inconvénients des prototypes

- ❖ Le prototype doit être pris en compte lorsqu'un système doit créer de nouveaux objets de nombreux types dans une hiérarchie de classes complexe.
- ❖ L'inconvénient de l'utilisation de Prototype est que faire une copie d'un objet peut parfois être compliqué.

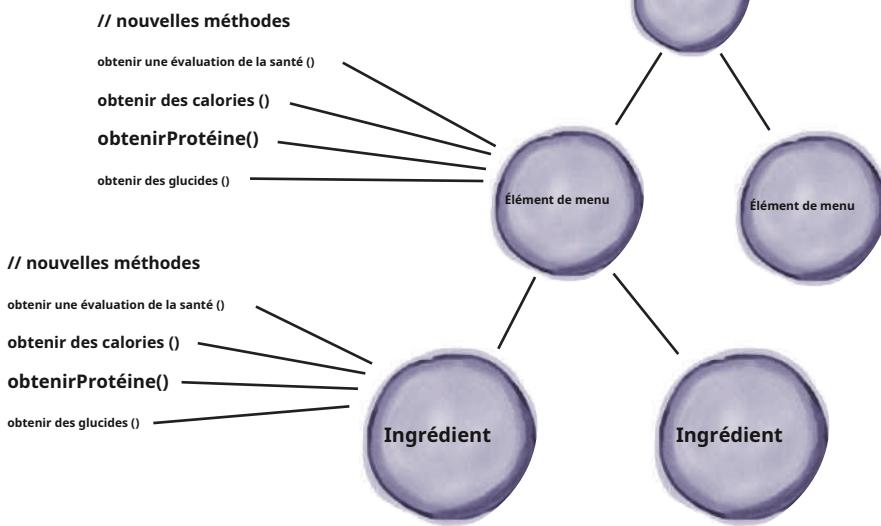
Visiteur

Utilisez le modèle Visiteur lorsque vous souhaitez ajouter des fonctionnalités à un composite d'objets et que l'encapsulation n'est pas importante.

Un scénario

Les clients qui fréquentent l'Objectville Diner et l'Objectville Pancake House sont récemment devenus plus soucieux de leur santé. Ils demandent des informations nutritionnelles avant de commander leurs repas. Comme les deux établissements sont très disposés à créer des commandes spéciales, certains clients demandent même des informations nutritionnelles pour chaque ingrédient.

La solution proposée par Lou :



Les inquiétudes de Mel...

« Mon garçon, on dirait qu'on ouvre la boîte de Pandore. Qui sait ce qui va se passer ?
« Nous allons devoir ajouter une nouvelle méthode ensuite, et chaque fois que nous ajoutons une nouvelle méthode, nous devons le faire à deux endroits. De plus, que se passe-t-il si nous voulons améliorer l'application de base avec, par exemple, une classe de recettes ? Nous devrons alors effectuer ces modifications à trois endroits différents... »

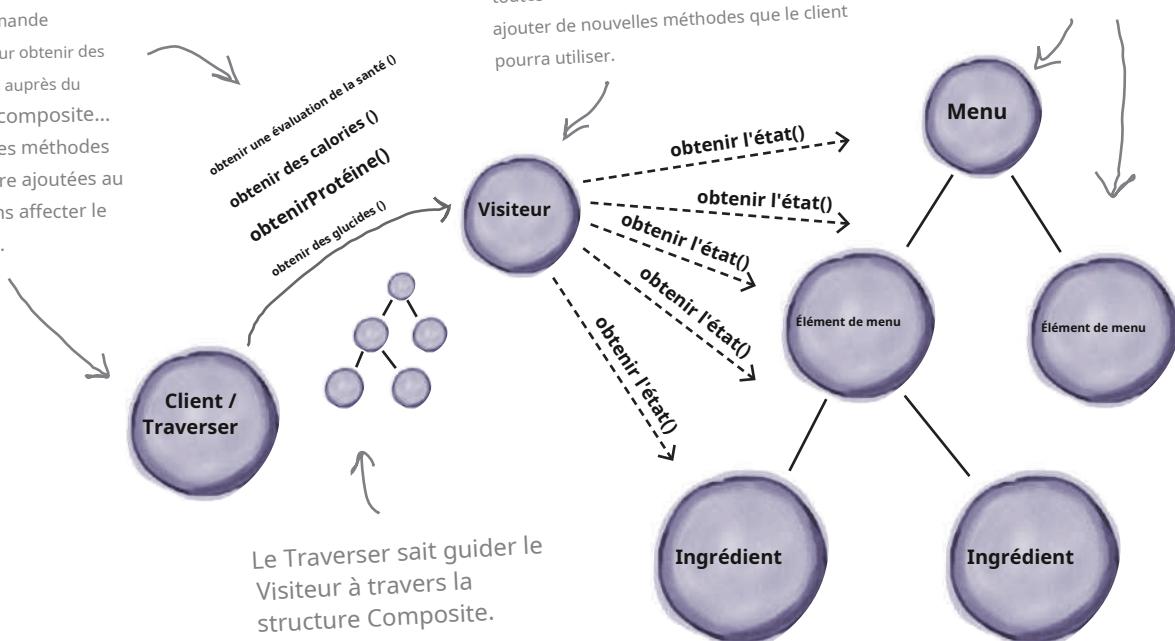
Le visiteur passe par ici

Le visiteur travaille en étroite collaboration avec un Traverser. Le Traverser sait comment naviguer vers tous les objets d'un Composite. Le Traverser guide le visiteur à travers le Composite afin que le visiteur puisse collecter l'état au fur et à mesure. Une fois l'état collecté, le client peut demander au visiteur d'effectuer diverses opérations sur l'état. Lorsqu'une nouvelle fonctionnalité est requise, seul le visiteur doit être amélioré.

Le client demande le visiteur pour obtenir des informations auprès du Structure composite... De nouvelles méthodes peuvent être ajoutées au visiteur sans affecter le Composite.

Le visiteur doit pouvoir appeler getState() dans toutes les classes, et c'est là que vous pouvez ajouter de nouvelles méthodes que le client pourra utiliser.

Tous ces composites les classes doivent ajouter une méthode getState() (et ne pas craindre de s'exposer).



Avantages pour les visiteurs

- ❖ Vous permet d'ajouter des opérations à une structure composite sans modifier la structure elle-même.
- ❖ L'ajout de nouvelles opérations est relativement facile.
- ❖ Le code des opérations effectuées par le Visiteur est centralisé.

Inconvénients pour les visiteurs

- ❖ L'encapsulation des classes composites est rompue lorsque le visiteur est utilisé.
- ❖ Étant donné que la fonction de traversée est impliquée, les modifications apportées à la structure composite sont plus difficiles.

UN

Classe AbstractButton 65 classe

abstraite 128, 292, 293

Modèle d'usine abstrait

environ 153

construction d'usines d'ingrédients 146–148, 167

combinaison de modèles 502–505, 548 définition
156–157

exercice de correspondance de description de
574, 596 Factory Method Pattern et 158–161

implémentant 158

entretien avec 158–159

RésuméListe 309

superclasses abstraites 12

Modèle d'adaptateur

environ 243–244

adaptation à l'interface Iterator Enumeration 251

combinaison de modèles 498–499

traitement de la méthode remove() 252

Modèle de décorateur vs. 254–255

défini 245

conception de l'adaptateur 251

exercices pour 256, 275, 375, 379, 481, 574, 596

Facade Pattern vs. 262

dans les adaptateurs d'objet et de classe

Modèle-Vue-Contrôleur 540 246–249

Modèle de proxy vs. 466

adaptateurs simples du monde réel 250

Écriture de l'adaptateur d'itérateur d'énumération 252–253

adaptateurs, OO (orienté objet)

environ 238–239

création d'adaptateurs bidirectionnels

244 en action 240–241

objet, classe objet et classe 246–249 essai
de conduite 242

agrégats 327, 338

Indice

couvertures d'album, affichage à l'aide d'un modèle proxy

environ 458

code pour 489–492

conception du proxy virtuel 459

processus de révision 465

visionneuse de test 464

écriture Image Proxy 460–463

Alexandre, Christophe

Un langage de modèles 588

La méthode intemporelle de construction du 588

algorithmes, encapsulation

environ 277

abstraction de prepareRecipe() 284–287

Modèle de stratégie et 24

Modèle de méthode de modèle et

environ 288–290

applets en 309

code de près 292–293

défini 291

Le principe d'Hollywood et les 298–300

accroches dans 293–295

dans le monde réel 301

tri avec 302–307

Balançoire et 308

code de test 296

algorithmes, famille de 22

Anti-modèles 592–593

Section Applicabilité, dans le catalogue de modèles

571 Modèles d'application 590

Modèles architecturaux 590 ArrayList,

tableaux et 320–325, 351

tableaux

itération et 325–326, 345 itérateur et

méthode hasNext() avec 328 itérateur et
méthode next() avec 328

B

catégories de modèles comportementaux, modèles de conception 576,
578–579

comportements

cours comme 14
classes étendues pour incorporer de nouvelles variables déclarantes 86 15
déléguer aux objets décorés tout en ajoutant 90 conception 11–12 encapsulant 11, 22 mise en œuvre 11, 13 séparer 10 réglage dynamique 20–21

Soyez les exercices de solution JVM, traitant du multithreading 179–180, 188

Modèle de pont 598–599

Modèles de construction 600–601

Modèles de processus métier 591

C

Proxy de mise en cache, en tant que forme de proxy virtuel 466, 482

Menu du café, intégration dans le framework (modèle Iterator) 347

Modèle de chaîne de responsabilité 602–603

changement

comme une constante dans le développement de logiciels 8 identifiant 54 itération et 340

exemple d'usine de chocolat, utilisant le modèle Singleton 175–176, 183

adaptateurs de classe, objet vs. 246–249 conception

de classe, du modèle Observer 51–52

classes. *Voir aussi* sous-classes

résumé 128, 292, 293 adaptateur 244, 274 Modèle d'adaptateur 245 modification du décorateur 108

comme comportements 14

collection 352

création de 10

étendu pour intégrer de nouveaux comportements 86

Créateur de modèle de méthode d'usine et produit 131–132 ayant une responsabilité unique 340–341 composant de haut niveau 139 identifiant comme classe Proxy 480 relations entre 22 états

définition 395

mise en œuvre 397, 400–405, 409

nombre croissant de remaniements dans 408

la conception 398–399

transitions d'état en 408 en utilisant la composition avec 23

utiliser des variables d'instance au lieu 82–83

d'utiliser des singletons statiques 184

Utilisation du nouvel opérateur pour instancier du béton 110–113

Section Classification, dans le catalogue de modèles 571

chargeurs de classes, utilisation avec Singletons 184 modèles

de classe, modèles de conception 577

tas client 433–436

assistant client (stubs), dans RMI 436–437, 440, 442–444, 453–454

Exercice Code Magnets

pour DinerMenu Iterator 354, 378 pour Observer Pattern 70, 76

cohésion 340

Section Collaborations, dans le catalogue de modèles 571

classes de collection 352

collection d'objets

abstraction avec le modèle Iterator

environ 317

ajout des itérateurs 328–334

nettoyage du code à l'aide de java.util.Iterator 335–337

méthode remove() dans 334

implémentation d'itérateurs pour 327

intégration dans le cadre 347

signification de 327

en utilisant un modèle composite

environ 363

implémentation des composants 364–366

code de test 368–370

structure arborescente 360–362, 368 en

utilisant les relations tout-partie 372

Collections, itérateurs et 353

Combinaison de motifs

Modèle d'usine abstraite 502–505
 Modèle d'adaptateur 498–499
 diagramme de classe pour 518–519
 Modèle composite 507–509
 Modèle décorateur 500–501
 Modèle itérateur 507
 Modèle d'observateur 510–516

objets de commande

encapsulation des requêtes pour faire quelque chose 196
 mappage 201
 en utilisant 204

Modèle de commande

objets de commande
 bâtiment 203
 encapsulation des requêtes pour faire quelque chose 196
 mappage 201
 en utilisant 204
défini 206–207
 objets de commande muets et intelligents 228
 exercice correspondant à la description de 574,
 596 télécommande domotique
 environ 193
 bâtiment 203–205, 235
 diagramme de classe 207
 création de commandes à charger 208–209
 définition 206
 conception 195–196
 Mise en œuvre de 210–212
 commandes macro 225, 226–228, 236
 mappage 201–202, 235
 Objet nul dans 214 tests
 204, 212–213, 227
 commandes d'annulation 217–221, 223–224, 228, 236 classes
 de fournisseurs pour 194
 rédaction de la documentation 215
 journalisation des requêtes à l'aide de 230
 cartographie 201–202, 235
Objet nul 214

mise en file d'attente des demandes à

l'aide de 229 compréhension 197–200

méthode compareTo() 303

Composants du proxy de masquage de
 complexité 483 de l'objet 267–271

Modèle composite

combinaison de modèles 507–509
 définis à 360°
 sous-menu dessert utilisant
 environ 357
 conception 363, 371
 Mise en œuvre de 364–367
tests 368–370

exercice correspondant à la description de 375, 379, 574, 596
 dans Modèle-Vue-Contrôleur 526–527, 543

entretien avec 372–373
 sur les questions de mise en œuvre 372–373
 sécurité et transparence 509
 transparence en 371
 structure arborescente de 360–362, 368

composition

ajout d'un comportement lors de
 l'exécution 85 favorisant l'héritage
 23, 85 héritage vs. 93
 adaptateurs d'objet et 249

motifs composés, utilisant

environ 493–494
Modèle-Vue-Contrôleur
 vers 520–521, 523–525
 Modèle d'adaptateur 539
 Modèle de rythme 529, 549–552
 Motif composite 526–527, 543
 Contrôleurs par vue 543
 Contrôleur cardiaque 541, 561 Modèle cardiaque
 539, 558–560 Mise en œuvre du contrôleur 536–537, 556–557 Mise en œuvre de DJ View 528–535,
 553–556 Modèle de médiateur 543

modèle en 543

Motif d'observation 526–527, 531–533,

chanson 520–521

état du modèle 543

Modèle de stratégie 526–527, 536–537, 539, 558–560
 tests 538

vues accédant aux méthodes d'état du modèle 543

plusieurs modèles vs. 516

cours concrets

dérivant de 143

Modèle d'usine et 134 se

débarrassant de 116

- instanciation d'objets et 138
en utilisant le nouvel opérateur pour instancier 110–113
variables contenant une référence à 143
- créateurs de béton 135
- objet d'implémentation concrète, affectation, 12
- méthodes concrètes, comme crochets 293–295 sous-classes concrètes 121–122, 297
- Section Conséquences, dans le catalogue de modèles 571
- contrôle de l'accès aux objets, à l'aide du modèle de proxy environ 426–428
- Proxy de mise en cache 466, 482 Proxy de masquage de complexité 483 Proxy de copie sur écriture 483 Proxy de pare-feu 482
- Procuration de protection environ 469
création d'un proxy dynamique 474–478 mise en œuvre d'un service de mise en relation 471–472 protection des sujets 473
service de jumelage de tests 479–480
Utilisation d'un proxy dynamique 469–470
- Proxy à distance environ 429
ajout au code de surveillance 432 préparation du service à distance 446–447 enregistrement auprès du registre RMI 448 réutilisation du client pour 449
processus de révision 453–455
rôle de 430–431
tests 450–452
objets d'emballage et 468
- Proxy de référence intelligent 482
- Proxy de synchronisation 483
- Proxy virtuel environ 457
conception du proxy virtuel 459
processus de révision 465
test 464
écriture Image Proxy 460–463
- Proxy de copie sur écriture 483
- créer une méthode
remplacement du nouvel opérateur par la méthode statique 116 par rapport à la méthode 115
en utilisant les sous-classes avec 121–122
- création de classes statiques au lieu de Singleton 179–180
- catégorie de modèles de création, modèles de conception 576, 578–579
- classes de créateur, dans le modèle de méthode d'usine 131–132, 134–135
- mots croisés 33, 74, 163, 187, 234, 273, 311, 374, 484**
- Cunningham, quartier 589
- ## D
- Modèle de décorateur environ 88–90, 104
Modèle d'adaptateur vs. 254–255
combinant les modèles 500–501
définis 91
inconvénients du 101
exercices pour 256, 275, 481, 574, 596
dans Java I/O 100–101
dans la catégorie Modèles structurels 577 entretien avec 104
Modèle de proxy par rapport au projet Starbuzz Coffee 466–468 environ 80–81
ajout de tailles au code 99 construction de commandes de boissons 89–90 dessin du processus de commande de boissons 94, 107 test du code de commande 98–99
code d'écriture 95–97
découplage, Itérateur permettant 333, 337, 339, 351–352
délégation, ajout de comportement à l'exécution 85 dépendance, dans le modèle d'observateur 52 Principe d'inversion de dépendance (DIP) 139–143, 300 pourriture de dépendance 298
- dépendre des abstractions principe de conception 139
- Modèles de conception devenir écrivain de 573
modèles de comportement catégorie 576, 578–579 catégories de 576–579 modèles de classe 577 modèles de création catégorie 576, 578–579 définis 565–567 découvrir son propre 572 exercice correspondant à la description de 596

- cadres vs. 29**
- guide pour une vie meilleure avec 564
 - implémentations sur interface dans 117
 - bibliothèques contre 29
 - modèles d'objets 577
 - organisation 575
 - utilisation excessive du 584
 - ressources pour 588–589
 - règle de trois appliquée à 573 modèles structurels catégories 576, 578–579 pensée en modèles 580–581
 - en utilisant 29, 582, 584 votre esprit sur les modèles 583
- Modèles de conception – Logiciels orientés objet réutilisables (Gamma et al.) 588**
- principes de conception
 - Principe d'inversion de dépendance 139–143
 - dépendant d'abstractions 139
 - Encapsuler ce qui varie 9, 73, 75, 136, 393 Privilégier la composition plutôt que l'héritage 23, 73, 75, 393 Le principe d'Hollywood 298–300
 - Une classe, une responsabilité, principes 184, 340, 371
 - un exemple. *Voir* Modèle Singleton Principe ouvert-fermé 355, 392 Principe de moindre connaissance 267–271
 - Programme vers une interface, pas une implémentation 11–12, 73, 75–76, 337
 - Principe de responsabilité unique (SRP) 340–341 en utilisant le modèle d'observateur 73, 75
- Puzzles de conception**
- dessin d'un diagramme de classe à l'aide de la vue et de la conception **Troll 536, 548**
 - dessin d'un ensemble parallèle de classes 133, 165 dessin d'un diagramme d'état 391, 420 de classes et d'interfaces 25, 34
 - repenser les classes pour supprimer la redondance 281, 278–283
 - refonte du sous-menu dessert Image Proxy 463, 486, à l'aide d'un modèle composite environ 357
 - conception 363, 371
 - Mise en œuvre de 364–367
 - tests 368–370**
 - menus de restaurant, fusion (modèle Iterator) environ 318–319
 - ajout des itérateurs 328–334
- nettoyage du code à l'aide de java.util.Iterator 335–337
 - encapsulation de l'itérateur 325–326
 - implémentation des itérateurs pour 327 implémentation de 320–325
 - DIP (Principe d'inversion de dépendance) 139–143, 300
 - Vue DJ 528–535, 549–561
 - Modèles spécifiques au domaine 590
 - verrouillage à double contrôle, réduisant l'utilisation de la synchronisation en utilisant 182
 - Exercices sur les aimants de canard, objet et classe objet et classe adaptateurs 247–248
 - Simulateur de canard, reconstruction environ 495–497
 - ajout du modèle d'usine abstraite 502–505, 548 ajout du modèle d'adaptateur 498–499
 - ajout du modèle composite 507–509
 - ajout du modèle décorateur 500–501
 - ajout du modèle itérateur 507
 - ajout du diagramme de classe Observer Pattern 510–516 518–519
 - objets de commande muets 228 aspect
 - dynamique des proxys dynamiques 480 proxy dynamique 469–470, 474–478

E

- Encapsuler ce qui varie principe de conception 9, 73, 75, 136, 393**
- encapsulant**
- comportement 11
 - code 22–23, 114–115, 136
 - itération 325–326
 - invocation de méthode 191, 206
 - construction d'objet 600
 - requêtes 206
- algorithmes d'encapsulation environ 277
- abstraction de prepareRecipe() 284–287 Modèle de méthode et environ 288–290
- RésuméListe et 309
- code de près 292–293
- défini 291
- Le principe d'Hollywood et 298–300

- crochets en 293–295
- dans le monde réel 301
- tri avec 302–307
- Balançoire et 308
- code de test 296
- sous-système d'encapsulation, Façades 262
- Énumération**
 - environ 250
 - s'adapter à l'itérateur 251
 - java.util Enumeration comme une ancienne implémentation de Itérateur 250, 342
 - méthode remove() et 252
 - Adaptateur d'écriture qui adapte l'itérateur à 253, 275
- exercices**
 - Soyez la solution JVM, gérez le multithreading 179–180, 188
- Codes aimantés**
 - pour DinerMenu Iterator 354, 378
 - pour les modèles d'observateurs 70, 76
- gérer le multithreading 247–248
- Puzzles de conception**
 - dessin d'un diagramme de classe à l'aide de la vue et contrôleur 536, 548
 - dessin d'un ensemble parallèle de classes 133, 165 dessin d'un diagramme d'état 391, 420 de classes et d'interfaces 25, 34
 - repenser les classes pour supprimer la redondance 281–282
 - refonte du proxy d'image 463, 486
- Exercices sur les aimants de canard, objet et objet de classe et adaptateurs de classe** 247
- implémentation de l'itérateur 329
- implémentation du bouton Annuler pour la macro-commande 228, 236
- Taillez votre crayon**
 - modification des classes de décorateur 99, 108 annotation des états de Gumball Machine 405, 423 annotation du diagramme d'état 396, 422
 - usine d'ingrédients de construction 148, 167 modification des classes pour le modèle de décorateur 512, 546 modification du code pour s'adapter au framework dans le modèle d'itérateur 347, 377
 - choisir des descriptions de l'état de mise en œuvre 392, 421
- diagramme de classe pour l'implémentation de prepareR-recette() 286, 314
- code n'utilisant pas les usines 137, 166 création de commandes pour les boutons d'arrêt 226, 236 création de l'indice de chaleur 62
- déterminer les classes violant le principe du moindre Connaissances 270, 274
- dessin du processus de commande de boissons 107 correction du code de la chaudière à chocolat 183, 190 identification des facteurs influençant la conception 84 mise en œuvre de la commande de porte de garage 205, 235 mise en œuvre des classes d'état 402, 421
- faire une pizzeria** 124, 164
- correspondance des modèles avec les catégories 575–577 méthode de remplissage du distributeur de boules de gomme 417, 424 sur l'ajout de comportements 14
- sur l'implémentation de printmenu() 324, 377
- sur l'héritage 5, 35
- esquisser des classes 55 éléments moteurs du changement 8, 35 transformer une classe en Singleton 176, 189 station météo SWAG 42, 75
- écrire un modèle d'usine abstraite 505, 548 écrire des classes pour les adaptateurs 244, 274 écrire un proxy dynamique 478, 487
- écriture du code d'observateur de troupeau 514, 547
- méthodes d'écriture pour les classes 83, 106 Qui fait quoi
 - faire correspondre les objets et les méthodes au modèle de commande sterne 202, 235
 - faire correspondre les modèles avec son intention 256, 275 faire correspondre le modèle avec la description 300, 314, 375, 379, 418, 424, 481, 488, 574, 596
- Adaptateur d'écriture qui adapte l'itérateur à l'énumération 253, 275
- gestionnaire d'écriture pour le service de matchmaking 477, 486
- itérateurs externes 342

F

Motif de façade

environ 256

Avantages du modèle d'adaptateur par rapport au modèle 262 262

- avantages du 262
- construire un système de cinéma maison
 - environ 257–259
 - construction de façade en 263 mise en œuvre de la classe de façade 260–262
 - implémentation de l'interface 264
- diagramme de classe 266
- Complexité masquant le proxy vs. 483** défini 266
- exercices pour 256, 275, 375, 379, 481, 574, 596
- Principe de la moindre connaissance et 271
- méthode d'usine**
 - environ 125, 134
 - comme résumé 135
 - déclarant 125–127
- Modèle de méthode d'usine
 - environ 131–132
 - à propos des objets d'usine 114
 - Modèle d'usine abstrait et code 158–161 de près 151
 - classes concrètes et 134
 - classes créatrices 131–132 déclaration de méthode de fabrique 125–127
 - définie 134
- Principe d'inversion de dépendance 139–143 dessin d'un ensemble parallèle de classes 133, 165
- exercice de correspondance de la description de 574, 596 entretien avec 158–159
- examen des dépendances des objets 138
- classes de produits 131–132
- Usine simple et 135**
- Modèle d'usine
- Usine abstraite**
 - environ 153
 - construction d'usines d'ingrédients 146–148, 167
 - combinaison de modèles 502–505, 548
 - définition 156–157
 - exercice correspondant à la description de 574, 596 Factory Method Pattern et 158–160
 - mise en œuvre de 158
 - exercice correspondant à la description de 300, 314
- Méthode d'usine**
 - environ 131–132
 - avantages du 135**
 - code de près 151
 - classes de créateurs 131–132
- déclaration de la méthode d'usine 125–127
- définie 134
- Principe d'inversion des dépendances 139–143
- dessiner un ensemble parallèle de classes 133, 165
- exercice correspondant à la description de 574, 596
- examiner les dépendances d'objets 138
- classes de produits 131–132
- Usine simple et 135**
- Usine simple**
 - à propos des objets d'usine 114
 - construction d'usine 115
 - défini 117**
 - Modèle de méthode d'usine et modèle 135 mention honorable 117
 - utiliser un nouvel opérateur pour instancier du béton Cours 110–113
- Privilégier la composition plutôt que l'héritage, principe de conception 23, 73, 75, 393
- Discussion au coin du feu
 - Modèle de décorateur contre modèle d'adaptateur 254–255
 - Modèle de stratégie contre modèle d'état 414–415
- Pare-feu proxy 482**
- Modèle de poids mouche 604–605
- forces 568
- pour la boucle 344
- frameworks vs bibliothèques 29

G

- Gamma, Erich 587–588
- Gang of Four (GoF) 569, 587–588
- point d'accès global 177
- variables globales, Singleton vs. 184
- guide pour mieux vivre avec les Design Patterns 564
- implémentation du contrôleur de machine à boules de gomme, utilisant l'état Modèle
 - nettoyage du code 413
 - démonstration du diagramme 411–412 au code 384–385
 - finition 410
- Concours un sur dix
 - environ 390–391
 - annotation du diagramme d'état 396, 422
 - modification du code 392–393

dessin d'un diagramme d'état 391, 420 implémentation de classes d'état 397, 400–405, 409 nouvelle conception 394–396

remaniement des classes d'état 398–399 Remplissage

de la machine à boules de gomme 416–417

SoldState et WinnerState dans le code de test 412 388–389
code d'écriture 386–387

Surveillance des distributeurs de chewing-gum, à l'aide de modèles de proxy

environ 426–428

Proxy à distance

environ 429

ajout au code de surveillance 432 préparation du service à distance 446–447 enregistrement auprès du registre RMI 448 réutilisation du client pour 449

processus de révision 453–454

rôle de 430–431

tests 450–452

objets d'emballage et 468

H

Relations HAS-A 23, 91 HashMap 348,

352, 353 méthode hasNext() 328, 342,

344 Principes d'apprentissage Head

First xxviii Helm, Richard 587–588

classes de composants de haut niveau

139 Le principe d'Hollywood 298–300

télécommande domotique, utilisant le modèle de commande

environ 193

bâtiment 203–205, 235

diagramme de classe 207

création de commandes à charger 208–209

définition 206

conception 195–196

Mise en œuvre de 210–212

commandes macro

environ 225

codage en dur contre bouton

d'annulation 228 228, 236

en utilisant 226–227

cartographie 201–202, 235

Objet nul 214

test des commandes d'annulation

204, 212–213, 227

création de 217–219, 228

implémentation pour la macro-commande

236 test 220, 223–224

utiliser l'état pour implémenter 221

classes de vendeurs pour 194

rédaction de la documentation du

système home cinéma 215, construction

environ 257–259

construction de façade en 263

implémentation de l'interface 264

Aiguisez votre crayon 270 en utilisant

le modèle de façade 260–262

crochets, dans le modèle de méthode Template 293–295

je

Image Proxy, écriture 460–463 implantations 13, 17, 43

section Implémentation, dans le catalogue de modèles 571

implémenter sur l'interface, dans les modèles de conception

117 instructions d'importation et de package 128

héritage

composition vs. 93

inconvénients de 5, 85 favorisant la composition par rapport à 23 pour

l'entretien 4

pour réutilisation 4, 13

mise en œuvre de plusieurs 246

variables d'instance 82–83, 97–98 instanciation 110–113, 138,

170–172 intégration du menu Café, à l'aide du modèle

Iterator 347 section Intention, dans le catalogue de modèles

571

interface 11–12, 110–113

type d'interface 15, 18

itérateurs internes 342

Modèle d'interprétation 606–607

Entretien avec

Modèle composite 372–373

Modèle de décorateur 104

Modèle de méthode d'usine et modèle d'usine abstrait 158–159
 Modèle Singleton 174
 inversion, dans le principe d'inversion de dépendance 141
 invocateur 201, 206–207, 209,
 233 Relations IS-A 23
 Interface itérable 343
 Modèle d'itérateur
 environ 327
 diagramme de classe 339
 code en gros plan avec HashMap 348
 code violant le principe d'ouverture-fermeture 355–356
 Recouvrements et 353
 combinaison de motifs 507
 défini 338–339
 exercice correspondant à la description de 375, 379, 574,
 596 intégrant le menu Café 347
 java.util.Iterator 334
 fusionner les menus des restaurants
 environ 318–319
 ajout des itérateurs 328–334
 nettoyage du code à l'aide de java.util.Iterator 335–337
 encapsulation de l'itérateur 325–326
 implémentation d'itérateurs pour 327
 mise en œuvre de 320–325
 suppression d'objets 334
 Principe de responsabilité unique (SRP) 340–341
 Itérateurs
 ajout de 328 à 334
 permettre le découplage 333, 337, 339, 351–352
 nettoyage du code à l'aide de java.util.Iterator 335–337
 Collections et 353
 encapsulation 325–326
 Dénombrement s'adaptant à 251, 342
 externe 342
 HashMap et 353
 mise en œuvre de 327
 interne 342
 commande de 342
 code polymorphe utilisant 338, 342
 utilisant ListIterator 342
 Adaptateur d'écriture pour l'énumération 252–253 Adaptateur d'écriture qui s'adapte à l'énumération 253, 275

J

Bibliothèque JavaBeans 65
 Cadre de collections Java 353
 Décorateurs Java (packages java.io) 100–103 Kit de développement Java (JDK) 65
 Interface itérable Java 343
 Paquet java.lang.reflect, prise en charge du proxy dans 440, 469, 476
 java.util.Collection 353
 java.util Enumeration, comme ancienne implémentation d'Iterator 250, 342
 java.util.Iterator
 nettoyage du code à l'aide de l'interface 335–337 de 334
 en utilisant 342
 Machines virtuelles Java (JVM) 182, 432
 Classe JButton 65
 Cadres J, Swing 308
 Johnson, Ralph 587–588

K

Keep It Simple (KISS), dans la section Utilisations connues des modèles 580, dans le catalogue de modèles 571

L

expressions lambda 67
 Loi de Déméter. Voir Principe de la moindre connaissance - instantiation paresseuse 177
 feuilles, dans la structure arborescente du modèle composite 360–362, 368
 bibliothèques 29
 Liste chaînée 352
 ListIterator 342
 journalisation des requêtes, en utilisant le modèle de commande 230 en parcourant les éléments du tableau 323
 Principe de couplage lâche 54

M

commandes macro

- environ 225
- commandes macro 228, 236
- utilisant 226–227

entretien, héritage pour, 4

service de jumelage, utilisant le modèle Proxy

- environ 470
- création d'un proxy dynamique 474–478
- implémentation 471–472
- protection des sujets 473
- tests 479–480

Modèles de médiateur 543, 608–609

Modèles de memento 610–611

fusionner les menus des restaurants (modèle Iterator)

- environ 318–319
- ajout des itérateurs 328–334
- nettoyage du code à l'aide de `java.util.Iterator` 335–337
- encapsulation de l'itérateur 325–326
- implémentation des itérateurs pour 327 implémentation de 320–325

méthode des objets, composants de l'objet vs. 267–271

méthodes 143, 293–295

état de modélisation 384–385

Modèle-Vue-Contrôleur (MVC)

- vers 520–521, 523–525
- Modèle d'adaptateur 540
- Modèle de rythme 529, 549–552
- Motif composite 526–527, 543
- Contrôleurs par vue 543
- Contrôleur cardiaque 541, 561
- Modèle cardiaque 539
- implémentation du contrôleur 536–537, 556–557
- implémentation de DJ View 528–535, 553–556
- Modèle de médiateur 543
- modèle en 543
- Motif d'observation 526–527, 531–533,
- chanson 520–521
- état du modèle 543
- Modèle de stratégie 526–527, 536–537, 539, 558–560
- tests 538
- vues accédant aux méthodes d'état du modèle 543

Section Motivation, dans le catalogue de modèles 571

plusieurs modèles, en utilisant

environ 493–494

dans le simulateur de canard

à propos de la reconstruction 495–497

ajout du modèle d'usine abstraite 502–505, 548 ajout du modèle d'adaptateur 498–499

ajout du modèle composite 507–509

ajout du modèle décorateur 500–501

ajout du modèle itérateur 507

ajout du diagramme de classe Observer

Pattern 510–516 518–519

multithreading 181–182, 188

N

Section Nom, dans le catalogue de modèles 571

nouvel opérateur

relatif au modèle Singleton 171–172

remplacé par la méthode concrète 116

méthode next() 328, 342, 344 NoCommand, dans le

code de contrôle à distance 214

nœuds, dans la structure arborescente du modèle composite 360–362, 368

Objets nuls 214

O

accès aux objets, en utilisant le modèle de proxy pour le contrôle

environ 426–428

Proxy de mise en cache 466, 482 Proxy

de masquage de complexité 483 Proxy

de copie sur écriture 483 Proxy de pare-feu 482

Procuration de protection

environ 469

création d'un proxy dynamique 474–478 mise en œuvre d'un service de mise en relation 471–472 protection des sujets 473

service de jumelage de tests 479–480

Utilisation d'un proxy dynamique 469–470

Proxy à distance

environ 429

ajout au code de surveillance 432 préparation du service à distance 446–447 enregistrement auprès du registre RMI 448

- réutilisation du client pour le processus de révision 449 rôle de 430–431
- tests 450–452**
- objets d'emballage et 468
- Proxy de référence intelligent 482
- Proxy de synchronisation 483
- Proxy virtuel
 - environ 457
 - conception du proxy virtuel 459
 - processus de révision 465
 - test 464**
 - écriture Image Proxy 460–463 Adaptateurs
 - d'objets et adaptateurs de classes 246–249
- construction d'objet, encapsulation 600 création
- d'objet, encapsulation 114–115, 136
- Conception orientée objet (OO). *Voir aussi* principes de conception
 - adaptateurs
 - environ 238–239
 - création d'adaptateurs bidirectionnels 244 en action 240–241, 242
 - objet et classe objet et classe 246–249 modèles de conception vs. 30–31
 - extensibilité et modification du code OS en 87
 - lignes directrices pour éviter la violation de l'*Inversibilité de Dépendance* Principe de la délimitation 143
 - conceptions faiblement couplées et 54
 - modèles d'objets, modèles de conception 577
- objets**
 - composants de 267–271
 - création de 134
 - conceptions faiblement couplées entre 54
 - partageant l'état 408
 - Singleton 171, 174**
 - emballage 88, 244, 254, 262, 502
- Modèle d'observateur
 - environ 37, 44
 - modèles de classe catégorie 574
 - combinaison de modèles 510–516
 - comparaison avec Publish-Subscribe 45
 - dépendance dans 52
 - exemples de 65
 - exercice correspondant à la description de 375, 379, 596
 - dans Five-minute drama 48–50
- dans Modèle-Vue-Contrôleur 526–527, 531–533
- couplage lâche dans 54
- Objet observateur dans 45 relations un-à-plusieurs 51–52 processus 46–47
- Objet sujet dans 45
- stations météo utilisant
 - éléments d'affichage de bâtiment 60
 - conception 57
 - mise en œuvre de 58
 - mise sous tension 61
 - SWAG 42**
- observateurs
 - dans le diagramme de classe 52
 - dans le drame de cinq minutes 48–50
 - dans le modèle Observer 45
- OCP (Principe Ouvert-Fermé) 355, 392
- Principe d'une classe, d'une responsabilité. *Voir* Ré-simple
 - Principe de responsabilité (SRP)
- un concours sur dix dans une machine à boules de gomme, en utilisant le modèle State environ 390–391
 - annotation du diagramme d'état 396, 422
 - modification du code 392–393
 - dessin d'un diagramme d'état 391, 420 implémentation de classes d'état 397, 400–405, 409 nouvelle conception 394–396
 - remaniement des classes d'état 398–399
- Conception OO (orientée objet). *Voir* Orienté objet (OO)
 - conception
 - Principe d'ouverture-fermeture (OCP) 355, 392 Modèles d'organisation 591
 - utilisation excessive des modèles de conception 584
- P**
- instructions de package et d'importation 128 Section
- Participants, dans le catalogue de modèles 571
- hiérarchie partie-tout 360
- catalogues de motifs 567, 569–572 Pattern
- Death Match pages 493 A Pattern
- Language (Alexander) 588 motifs, utilisation de composés 493–494

- modèles, en utilisant plusieurs environ 493 dans le simulateur de canard à propos de la reconstruction 495–497 ajout du modèle d'usine abstraite 502–505, 548 ajout du modèle d'adaptateur 498–499 ajout du modèle composite 507–509 ajout du modèle décorateur 500–501 ajout du modèle itérateur 507 ajout du diagramme de classe Observer Pattern 510–516 518–519 modèles de patrons, utilisations de 573 Projet de pizzeria, utilisant le modèle Factory Résumé Usine en 153, 156–157 coulisses 154–155 construction de l'usine 115 sous-classes concrètes dans 121–122 dessin d'un ensemble parallèle de classes 133, 165 encapsulation de la création d'objets 114–115 garantie de la cohérence des ingrédients 144–148, 167 cadre pour 120 magasin franchisé 118–119 identification des aspects 112–113 mise en œuvre 142 faire une pizzeria en 123–124 commander une pizza 128–132 référencement des usines d'ingrédients locaux 152 pizzas retravaillées 149–151 code polymorphe, utilisant l'itérateur 338, 342 polymorphisme 12 prepareRecipe(), abstraction 284–287 Principe de la moindre connaissance 267–271. *Voir aussi* Célibataire Principe de Responsabilité (SRP) méthode print(), dans le sous-menu dessert en utilisant Composite Paternoster 364–367 programmation 12 Programme vers une interface, pas une conception d'implémentation principes 11–12, 73, 75–76, 337 programme vers interface vs programme vers supertype 12 Procuration de protection environ 469 création d'un proxy dynamique 474–478 implémentation d'un service de mise en relation 471–472 protection des sujets 473 Proxy Pattern et 466 test du service de mise en relation 479–480 à l'aide d'un proxy dynamique 469–470 Prototype de modèle 612–613 proxys 425 Classe proxy, identifiant la classe comme 480 Modèle de proxy Modèle d'adaptateur contre 466 Proxy de masquage de complexité 483 Proxy de copie sur écriture 483 Modèle de décorateur contre 466–468 défini 455–456 aspect dynamique des proxys dynamiques 480 exercice correspondant à la description de 481, 574, 596 Firewall Proxy 482 implémentation du proxy distant environ 429 ajout au code de surveillance 432 préparation du service à distance 446–447 enregistrement auprès du registre RMI 448 réutilisation du client pour 449 processus de révision 453–454 rôle de 430–431 tests 450–452 objets d'emballage et 468 Paquets java.lang.reflect 440, 469, 476 Proxy de protection et environ 469 Adaptateurs et 466 création d'un proxy dynamique 474–478 mise en œuvre d'un service de mise en relation 471–472 protection des sujets 473 service de jumelage de tests 479–480 Utilisation d'un proxy dynamique 469–470 Sujet réel en tant que substitut de 466 invoquant la méthode sur 475 obliger le client à utiliser le proxy au lieu du 466 en passant dans le constructeur 476 renvoyer un proxy pour 478 restrictions sur le passage des types d'interfaces 480 Proxy de référence intelligent 482 Proxy de synchronisation 483 variantes 466, 482–483

Proxy virtuel

- environ 457
- Proxy de mise en cache sous forme de 466, 482
- conception 459
- processus de révision 465
- test 464**
- écriture Image Proxy 460–463 Publier-S'abonner, en tant que modèle d'observateur 45

Q

Mise en file d'attente des demandes, à l'aide du modèle de commande 229

R

Sujet réel

- en tant que substitut du modèle de proxy
- 466 invoquant la méthode sur 475
- obliger le client à utiliser un proxy au lieu de 466 en passant dans le constructeur 476
- envoyer un proxy pour 478
- refactorisation 358, 581
- Section des modèles associés, dans le catalogue de modèles 571 relations, entre les classes 22
- Invocation de méthode à distance (RMI)
 - environ 432–433, 436
 - code de près 442
 - compléter le code côté serveur 441–444
 - importer java.rmi 446
 - importation de paquets 447, 449
 - création d'un service distant 437–441
 - appel de méthode dans 434–435
 - inscription au registre RMI 448
 - choses à surveiller dans 444

Proxy à distance

- environ 429
- ajout au code de surveillance 432 préparation du service à distance 446–447 enregistrement auprès du registre RMI 448 réutilisation du client pour 449
- processus de révision 453–454
- rôle de 430–431
- tests 450–452**
- objets d'emballage et 468

méthode remove()

- Dénombrement et 252
- dans la collection d'objets 334
- dans java.util.Iterator 342
- requêtes, encapsulation de 206 ressources, modèles de conception 588–589 réutilisation 4, 85
- règle de trois, appliquée à l'invention de modèles de conception 573 erreurs d'exécution, causes de 135

S

- Section de code d'exemple, dans le catalogue de modèles 571, tas de serveur 433–436
- aide de service (squelettes), dans RMI 436–437, 440, 442–444, 453–454
- vocabulaire partagé 26–27, 28, 585–586
- Taillez votre crayon**
 - modification des classes de décorateur 99, 108 annotation des états de Gumball Machine 405, 423 annotation du diagramme d'état 396, 422
 - usine d'ingrédients de construction 148, 167, changement de classes pour le modèle de décorateur 512, 546
 - modification du code pour l'adapter au framework dans le modèle Iterator 347, 377
- choisir des descriptions de l'état de mise en œuvre 392, 421
- diagramme de classe pour l'implémentation de prepareRecipe() 286, 314
- code n'utilisant pas les usines 137, 166 création de commandes pour les boutons d'arrêt 226, 236 création de l'indice de chaleur 62
- déterminer les classes violant le principe du moindre Connaissances 270, 274
- dessin du processus de commande de boissons 107 correction du code de la chaudière à chocolat 183, 190 identification des facteurs influençant la conception 84 mise en œuvre de la commande de porte de garage 205, 235 mise en œuvre des classes d'état 402, 421
- faire une pizzeria 124, 164**
 - correspondance des modèles avec les catégories 575–577 méthode de remplissage du distributeur de boules de gomme 417, 424 sur l'ajout de comportements 14
 - sur l'implémentation de printmenu() 324, 377

- sur l'héritage 5, 35 esquisser des classes 55
éléments moteurs du changement 8, 35
transformer une classe en Singleton 176, 189
station météo SWAG 42, 75 écriture d'un modèle d'usine abstraite 505, 548 écriture de classes pour les adaptateurs 244, 274 écriture d'un proxy dynamique 478, 487
- écriture du code d'observateur de troupeau 514, 547 méthodes d'écriture pour les classes 83, 106
- Modèle d'usine simple**
à propos des objets d'usine 114
construction d'usine 115
définition de 117
Modèle de méthode d'usine et modèle 135 mention honorable 117
utiliser un nouvel opérateur pour instancier des classes concrètes 110–113
- Principe de responsabilité unique (SRP) 184, 340–341, 371
- Objets singleton 171, 174
- Modèle Singleton**
environ 169–172
avantages du 170
Chocolaterie 175–176, 183
diagramme de classe 177
code de près 173
gestion du multithreading 179–182, 188
défini 177
inconvénients du 184
exercice de verrouillage doublement vérifié 182
correspondant à la description de 574 variables globales contre 184
mise en œuvre de 173
entretien avec 174
- Principe d'une classe, d'une responsabilité et 184 sous-classes dans 184
en utilisant 184
- squelettes (aides de service), dans RMI 436–437, 440, 442–444, 453–454
- objets de commande intelligents 228
- Proxy de référence intelligent 482
- développement de logiciels, changement en tant que constante dans 8
- méthodes de tri, dans la méthode Template Method Pattern 302–307
- sort() méthode 306–311
- méthode du séparateur 343
- Principe de responsabilité unique (PRS) 184, 340–341, 371
- Projet de manuel de formation Starbuzz Coffee Barista environ 278–283
abstraction de prepareRecipe() 284–287 à l'aide de la méthode Template Pattern environ 288–290
code de près 292–293
défini 291
Le principe d'Hollywood et les 298–300 accroches dans 293–295
code de test 296
- Projet Starbuzz Coffee, utilisant le modèle Decorator environ 80–81
ajout de tailles au code 99 construction de commandes de boissons 89–90 dessin du processus de commande de boissons 94, 107 test du code de commande 98–99
- code d'écriture 95–97**
machines d'état 384–385
- Modèle d'état**
défini 406
exercice correspondant à la description de l'implémentation du contrôleur de machine à boules de gomme 418, 424, 574, 596
nettoyage du code 413
démonstration du diagramme 411–412 au code 384–385 finition 410
remplissage du distributeur de boules de gomme 416–417 SoldState et WinnerState dans le code de test 412 388–389
code d'écriture 386–387
nombre croissant de classes dans la conception 408 état de modélisation 384–385
- Concours un sur dix dans un distributeur de chewing-gum environ 390–391
annotation du diagramme d'état 396, 422
modification du code 392–393
dessin d'un diagramme d'état 391, 420 implémentation de classes d'état 397, 400–405, 409 nouvelle conception 394–396
remaniement des classes d'état 398–399 partage d'objets d'état 408
transitions d'état dans les classes d'état 408
- Modèle de stratégie vs. 381, 407, 414–415
transitions d'état, dans les classes d'état 408

état, utilisé pour implémenter des commandes d'annulation 221 classes statiques, utilisant à la place des singlets 184 méthode statique vs. méthode de création 115 Modèle de stratégie algorithmes et 24 défini 24 exercice correspondant à la description de 300, 314, 375, 379, 418, 424, 574, 596 dans Modèle-Vue-Contrôleur 526–527, 536–537, 539 Modèle d'état vs. 381, 407, 414–415 Modèle de méthode Pattern et 307 s'efforcer d'obtenir des conceptions faiblement couplées entre les objets qui principe de conception teract 54. *Voir aussi* Principe de couplage lâche catégorie de modèles structurels, modèles de conception 576, 578–579 Section de structure, dans le catalogue de modèles 571 stubs (assistant client), dans RMI 436–437, 440, 442–444, 453–454 sous-classes commandes concrètes et 207 états concrets et 406 explosion de classes 81 Méthode d'usine et laisser les sous-classes décider lesquelles classe à instancier 134 dans Singleton 184 Magasin de pizzas en béton 121–122 Modèle Méthode 288 dépannage 4 Sujet dans le diagramme de classe 52 dans le drame de cinq minutes 48–50 dans le modèle Observer 45–47 sous-systèmes, façades et 262 superclasses 4, 12 supertype (programmation vers interface), vs. programmation vers interface 12 Synchronisation de la bibliothèque Swing 65, 308, 543, comme surcharge 180 Proxy de synchronisation 483

T

Modèle de méthode de modèle environ 288–290 classe abstraite en 292, 293, 297 applets en 309 diagramme de classe 291 code de près 292–293 défini 291 exercice correspondant à la description de 300, 314, 418, 424, 574, 596 Le principe d'Hollywood et les accroches 298–300 dans 293–295, 297 dans le monde réel 301 tri avec 302–307 Modèle de stratégie et 307 Swing et 308 code de test 296 pensée en modèles 580–581 étroitement couplée 54 La manière intemporelle de construire (Alexander) 588 transparence, dans le modèle composite 371 structure arborescente, modèle composite 360–362, 368 adaptateurs bidirectionnels, création de 244 tapez les paramètres sécurisés 135

Tu

annuler les commandes création de 217–219, 228 mise en œuvre de la prise en charge de la macro-commande 228 de 217 tests 220, 223–224 utiliser l'état pour implémenter 221 Modèles de conception d'interface utilisateur 591

V

variables comportement déclarant 15 contenant une référence à la classe concrète 143, instance 82–83, 97–98

Vecteur 352
Proxy virtuel
 environ 457
 Proxy de mise en cache comme forme 466, 482
 de conception 459
 processus de révision 465
 test 464
 écriture Image Proxy 460–463
Modèle de visiteur 614–
615 Vlissides, John 587–
588 mot-clé volatile 182

L

station météo

 éléments d'affichage de bâtiment 60
 conception 57
 mise en œuvre de 58
 mise sous tension 61

Exercices Qui fait quoi

 faire correspondre les objets et les méthodes au modèle de commande 202, 235
 faire correspondre les modèles avec son intention 256, 275
 modèle correspondant avec description 300, 314, 375, 379, 418, 424, 481, 488, 574, 596
 relations tout-partie, collection d'objets utilisant le site Web 372 Wickedlysmart xxxiii
 emballage des objets 88, 244, 254, 262, 468, 502

Y

votre esprit sur les modèles de conception modèle 583



Vous ne connaissez pas le site ?

Nous avons des mises à jour, des vidéos, des
projets, des blogs d'auteurs et plus encore !

**Apportez votre cerveau sur
wickedlysmart.com**



O'REILLY®

Il y a bien plus d'où cela vient.

Découvrez des livres, des vidéos, des cours de formation en ligne en direct et bien plus encore d'O'Reilly et de nos plus de 200 partenaires, le tout au même endroit.

Pour en savoir plus, rendez-vous sur oreilly.com/online-learning