Analyse et conception orientées objet

Tête la première

Analyse et conception orientées objet

Tête la première

Brett D. McLaughlin

Gary Pollice

David West

Traduction de

Sophie Govaere

Analyse et conception orientées objet — Tête la première

de Brett D. McLaughlin, Gary Pollice et David West

© 2007 O'Reilly Media et 2009 Digit Books pour la traduction française

ISBN : 978-2-8150-0004-8

Conception :	**Kathy Sierra, Bert Bates**
Responsable d'édition VO :	**Brett D. McLaughlin**
Éditeur VO :	**Mary O'Brien**
Éditrice VF :	**Dominique Buraud**
Couverture conçue par :	**Mike Kohnke, Edie Freedman**
OO :	**Brett D. McLaughlin**
A :	**David West**
C :	**Gary Pollice**

Les programmes figurant dans ce livre ont pour but d'illustrer les sujets traités. Il n'est donné aucune garantie quant à leur fonctionnement une fois compilés, assemblés ou interprétés dans le cadre d'une utilisation professionnelle ou commerciale.

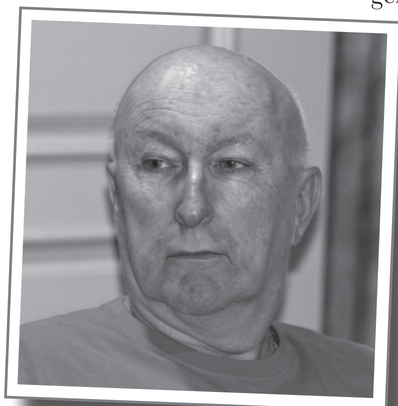
Toute représentation ou reproduction, intégrale ou partielle, faite sans le consentement de l'auteur, de ses ayants droit, ou ayants cause, est illicite (loi du 11 mars 1957, alinéa 1er de l'article 40). Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait une contrefaçon sanctionnée par les articles 425 et suivants du Code pénal. La loi du 11 mars 1957 autorise uniquement, aux termes des alinéas 2 et 3 de l'article 41, les copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective d'une part et, d'autre part, les analyses et les courtes citations dans un but d'exemple et d'illustration.

À tous ces gens géniaux qui ont imaginé différentes façons de rassembler des exigences, d'analyser un logiciel et de concevoir du code...

...merci d'avoir inventé quelque chose d'assez bon pour produire du bon logiciel, mais d'assez complexe pour que nous ayons besoin d'écrire ce livre pour tout expliquer.

Brett McLaughlin est un guitariste qui peine encore à accepter l'idée qu'on ne peut pas payer ses factures en jouant du blues et du jazz acoustique. Il a récemment découvert, à sa grande joie, qu'écrire des livres pour aider les gens à devenir de meilleurs programmeurs peut payer les factures. Il en est très content, ainsi que sa femme Leigh et ses enfants, Dean et Robbie.

Avant de s'aventurer au pays de Tête la première, Brett développait des applications Java pour Nextel Communications et Allegiance Telecom, jusqu'à ce que cela devienne banal. Puis il s'est mis aux serveurs d'applications et a travaillé sur les aspects internes du conteneur EJB et moteur de servlet Lutris Enhydra. En passant, Brett s'est pris d'intérêt pour le logiciel *Open Source* et a participé à la création de plusieurs outils de programmation sympas comme Jakarta Turbine et JDOM. Vous pouvez lui écrire à *brett@oreilly.com*.



Gary Pollice se définit comme un vieil ours atrabilaire (un homme généralement vieux et colérique) qui a passé plus de 35 ans dans le métier à essayer de trouver ce qu'il voulait faire quand il serait grand. Bien qu'il n'ait pas encore grandi, il a finalement pris la décision, en 2003, de s'installer dans le sanctuaire de l'université où il corrompt les esprits des futures générations de développeurs logiciel avec des idées radicales comme « développez du logiciel pour vos clients, apprenez à travailler en équipe, la qualité, l'élégance et l'exactitude de la conception et du code sont importantes, et ce n'est pas un problème d'être un intello asocial tant que vous êtes un intello asocial génial. »

Gary est professeur de pratique (comprenez par là qu'il avait un vrai travail avant de devenir professeur) au Worcester Polytechnic Institute. Il vit dans le centre du Massachusetts avec sa femme, Vikki et leurs deux chiens, Aloysius et Ignatius. Vous pouvez aller voir sa page à l'adresse **`http://web.cs.wpi.edu/~gpollice/`**. N'hésitez pas à lui envoyer un message, que ce soit pour vous plaindre ou pour le féliciter de ce livre.

Dave West aime se décrire comme le roi des geeks. Malheureusement personne d'autre n'est d'accord avec cette description. On parle plutôt de lui comme d'un Anglais professionnel qui aime parler des meilleures pratiques de développement logiciel avec la passion et l'énergie d'un prêtre évangéliste. Dave travaille depuis peu pour Ivar Jacobson Consulting, où il est responsable du continent américain et peut satisfaire son désir de parler de développement logiciel, parler de rugby et de football et soutenir que le cricket est plus intéressant que le baseball.

Avant d'être responsable du continent américain pour Ivar Jacobson Consulting, Dave a travaillé un certain nombre d'années à Rational Software (qui fait maintenant partie d'IBM). Dave a occupé plusieurs postes à Rational puis à IBM, dont responsable de produit pour RUP où il a introduit l'idée de procédures *plug-ins* et d'agilité. Vous pouvez lui écrire à dwest@ivarjacobson.com.

Table des matières (version courte)

	Intro	xxiii
1	Un bon logiciel commence ici : *une appli bien conçue qui assure*	1
2	Donnez-leur ce qu'ils veulent : *recueillir les exigences*	55
3	Je t'adore, tu es parfait... Mais maintenant, change : *les exigences changent*	111
4	Introduire votre logiciel dans le monde réel : *analyse*	145
5	1ère partie : Rien ne reste jamais comme avant : *bonne conception*	197
	Entracte : CATASTROPHE OO	221
	2ème partie : 30 minutes de gym pour votre logiciel : *logiciel souple*	233
6	« Je m'appelle Numérobis » : *résoudre de très gros problèmes*	279
7	Mettre de l'ordre dans le chaos : *architecture*	323
8	On accorde trop d'importance à l'originalité : *principes de conception*	375
9	Le logiciel reste destiné au client : *itérer et tester*	423
10	Tout assembler : *le cycle de vie acoo*	483
	Annexe A : *les restes*	557
	Annexe B : *bienvenue à objectville*	575

Table des matières (version longue)

Intro

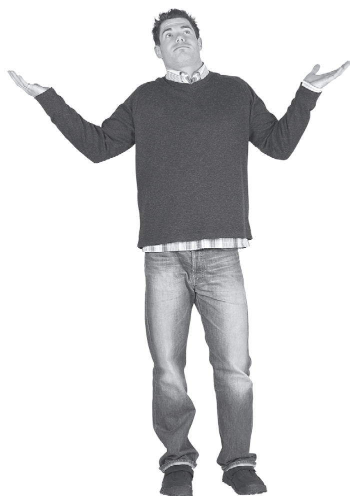
Votre cerveau et l'ACOO. Voilà que vous essayez d'apprendre quelque chose et que votre cerveau fait tout ce qu'il peut pour vous empêcher de le mémoriser. Votre cerveau pense : « Mieux vaut laisser de la place pour les choses vraiment importantes, comme savoir quels sont les animaux sauvages qu'il vaut mieux éviter ou que faire du snowboard en short n'est pas une bonne idée. ». Mais comment procéder pour lui faire croire que votre vie dépend de vos connaissances en analyse et conception orientées objet ?

A qui s'adresse ce livre ?	xxiv
Nous savons ce que vous pensez	xxv
La métacognition	xxvii
Domptez votre cerveau	xxix
Lisez-moi	xxx
L'équipe technique	xxxii
Remerciements	xxxiii

1 une appli bien conçue qui assure

Un bon logiciel commence ici

Alors comment écrit-on *vraiment* un bon logiciel ? Il n'est jamais facile de décider **par où commencer.** Est-ce que l'application fait effectivement **ce qu'elle est censée faire** ? Et que dire du code dupliqué ? Est-ce mauvais signe ? Il est toujours difficile de choisir **sur quoi travailler en premier,** tout en s'assurant de ne pas compromettre le reste de l'opération. Mais pas de souci. Quand vous aurez fini ce chapitre, vous **saurez comment écrire un bon logiciel** et vous aurez fait un grand pas en avant pour améliorer définitivement le développement de vos applications. Pour finir, vous comprendrez pourquoi vous avez *tout intérêt* à connaître l'ACOO.



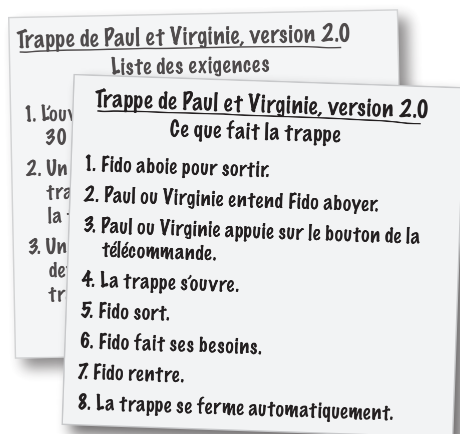
Le Rock 'n roll, c'est pour la vie !	2
La toute nouvelle application d'Éric	3
Quelle est la PREMIÈRE chose que vous changeriez ?	8
Un bon logiciel, c'est...	10
Un bon logiciels en 3 étapes faciles	13
D'abord, s'occuper de la fonctionnalité	18
Test	23
À la recherche de problèmes	25
Analyse	26
Appliquer des principes OO de base	31
Concevoir une fois, concevoir deux fois	36
Est-il facile de modifier vos applications ?	38
Encapsulez ce qui risque de changer	41
La délégation	43
Un bon logiciel, enfin (pour l'instant)	46
L'ACOO sert à écrire du bon logiciel	49
Points d'impact	50

recueillir les exigences

2 Donnez-leur ce qu'ils veulent

Tout le monde aime un client satisfait. Vous savez déjà que la première étape pour écrire du bon logiciel est de s'assurer qu'il fait ce que souhaite le client. Mais comment comprendre **ce qu'un client veut vraiment ?** Comment, d'ailleurs, s'assurer qu'un client *sait* vraiment ce qu'il veut ? C'est là qu'entrent en jeu les **exigences réelles** et dans ce chapitre, vous allez apprendre à **satisfaire votre client** en livrant, avec certitude, ce qu'il a demandé. Quand vous aurez fini, tous vos projets seront « satisfaction garantie » et vous aurez fait un pas de plus pour apprendre à écrire systématiquement du bon logiciel.

Vous avez un nouveau boulot de programmation	56
Test	59
Mauvaise utilisation (en quelque sorte)	61
Qu'est-ce qu'une exigence ?	62
Créer une liste d'exigences	64
Prévoir ce qui peut mal tourner	68
Les chemins alternatifs s'occupent des problèmes du système	70
Présentation des cas d'utilisation	72
Un cas d'utilisation, trois parties	74
Confronter vos exigences et vos cas d'utilisation	78
Votre système doit fonctionner dans le monde réel	85
Faisons connaissance avec le Chemin Heureux	92
Boîte à outils ACOO	106



La trappe du chien et la télécommande font partie du système (ou sont à l'intérieur du système).

3 les exigences changent

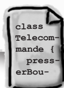
Je t'adore, tu es parfait... Mais maintenant, change !

Vous pensez avoir fait exactement ce que voulait le client ? Pas si vite... Vous avez discuté avec votre client, listé ses exigences, écrit vos cas d'utilisation et livré une application extra. C'est le moment de vous reposer devant un sympathique petit cocktail... jusqu'à ce que votre client décide qu'en fin de compte, il veut quelque chose de **différent. Pas ce qu'il vous avait dit**. Il adore ce que vous avez fait, vraiment, mais voilà, **ce n'est pas suffisant**. Dans le monde réel, **les exigences changent constamment** et c'est à vous de vous adapter pour que le client reste satisfait.

Vous êtes un héro !	112
Vous êtes une bille !	113
La constante en conception et analyse de logiciel	115
Chemin optionnel ? Chemin alternatif ? Qui peut les identifier ?	120
Les cas d'utilisation doivent être clairs	122
Du début à la fin : un scénario unique	124
Les confessions du Chemin Alternatif	126
Compléter la liste des exigences	130
Dupliquer du code est une mauvaise idée	138
Un dernier test	140
Écrivez votre propre principe de conception	141
Boîte à outils	142


```
public void presserBouton() {
  System.out.println("Quelqu'un appuie sur le bouton de la
télécommande...");
  if (trappe.isOuverte()) {
    trappe.fermer();
  } else {
    trappe.ouvrir();

    final Timer timer = new Timer();
    timer.schedule(new TimerTask() {
      public void run() {
        trappe.fermer();
        timer.cancel();
      }
    }, 60000);
  }
}
```



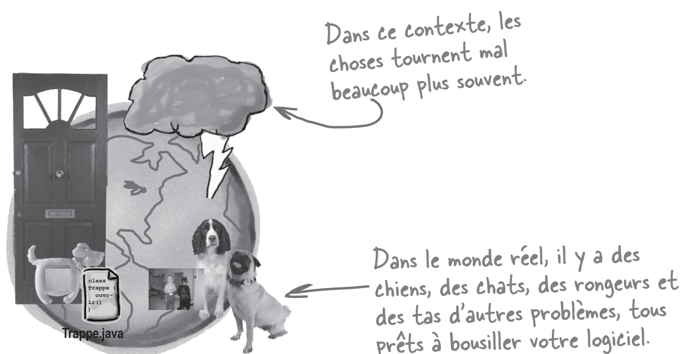
Telecommande.java

analyse

4 Introduire votre logiciel dans le monde réel

Il est temps de faire passer le test du monde réel à vos applications. Que votre application fonctionne sur votre machine, parfaitement installée et réglée au millimètre, c'est bien, mais elle doit faire mieux ; elle doit fonctionner quand **monsieur « tout le monde » l'utilise**. Ce chapitre va s'assurer que votre application fonctionne dans le **contexte du monde réel**. Vous verrez comment **l'analyse textuelle** peut transformer le cas d'utilisation sur lequel vous avez travaillé en classes et en méthodes qui, comme chacun sait, sont ce que veulent les clients. Et quand vous aurez fini, vous aussi, vous pourrez dire :
« Ça y est ! Mon logiciel est **prêt pour le monde réel** ! »

Un chien, deux, trois, quatre chiens, cinq, six... 146
Votre logiciel a un contexte 147
Identifier le problème 148
Prévoir une solution 149
L'histoire des deux codeurs 156
Détour par la délégation 160
Le pouvoir des applications lâchement couplées 162
Faites attention aux noms dans votre cas d'utilisation 167
D'une bonne analyse à de bonnes classes... 180
Les diagrammes de classe disséqués 182
Les diagrammes de classe ne sont pas suffisants 187
Points d'impact 191



Le monde réel

5 (1ère partie)

bonne conception = logiciel souple

Rien ne reste jamais comme avant

Le changement est inévitable. Vous adorez votre logiciel pour l'instant, mais demain il en sera peut-être autrement. Par ailleurs, plus votre logiciel sera difficile à modifier, plus il sera difficile de répondre aux besoins du client, qui changent constamment. Dans ce chapitre, nous allons rendre visite à un vieil ami, essayer d'améliorer un projet de logiciel existant et voir comment de petits changements peuvent se transformer en gros problèmes. En fait, nous allons découvrir un problème si gros qu'il nous faudra DEUX PARTIES pour le résoudre !

Les Guitares d'Éric s'agrandit 198
Classes abstraites 201
Les diagrammes de classe disséqués (2) 206
Antisèche UML 207
Tuyaux sur les problèmes de conception 213
Les 3 étapes pour un bon logiciel (revisitées) 215

5 (entracte)

CATASTROPHE OO !

Le jeu télévisé préféré d'Objectville

Pour éviter les risques	Concepteurs célèbres	Briques de code	Maintenance et réutilisation	Névroses logicielles
100 €	100 €	100 €	100 €	100 €
200 €	200 €	200 €	200 €	200 €
300 €	300 €	300 €	300 €	300 €
400 €	400 €	400 €	400 €	400 €

5 (2ème partie)

bonne conception = logiciel souple

30 minutes de gym pour votre logiciel

Aimeriez-vous être un peu plus souple ? Si vous vous heurtez à des problèmes en modifiant votre application, cela veut probablement dire que votre logiciel a besoin de **plus de souplesse et de résistance**. Pour aider votre application à s'étirer, vous allez faire de l'analyse, un bon paquet de conception et apprendre comment des principes OO peuvent vraiment débloquer votre application. Et pour terminer, vous verrez comment une plus grande **cohésion peut améliorer votre couplage**. Cela vous tente ? Tournez la page et retournons réparer cette application rigide.

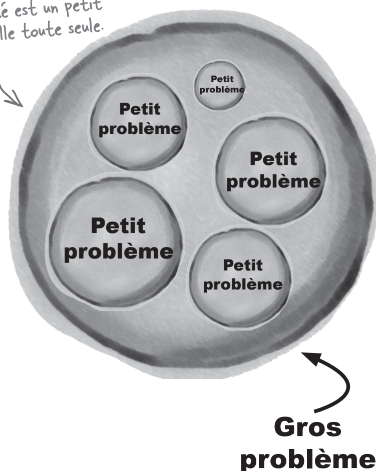
Retour à l'outil de recherche d'Éric 234
La méthode chercher() vue de plus près 237
Les avantages de notre analyse 238
Les classes sont axées sur le comportement 241
Mort d'une (décision de) conception 246
Changer de mauvaises décisions de conception en de bonnes décisions 247
La « double encapsulation » dans le logiciel d'Éric 249
N'ayez jamais peur de faire des erreurs 255
Admirez la souplesse de l'application d'Éric 258
Testons le logiciel bien conçu 261
Le logiciel d'Éric est-il facile à modifier ? 265
Le grand défi de la facilité-de-changement 266
Une classe cohésive fait une seule chose très bien 269
Le cycle de vie de la conception (cohésion) 272
Un bon logiciel doit être « suffisamment bon » 274
Boîte à outils ACOO 276

6 résoudre de très gros problèmes

« Je m’appelle Numérobis... Je suis architecte »

Il est temps de construire quelque chose de VRAIMENT GROS. Êtes-vous prêts ? Vous avez de nombreux outils dans votre boîte à outils ACOO, mais comment les utiliser pour construire quelque chose de **vraiment important** ? Vous ne le réalisez peut-être pas mais **vous avez tout ce dont vous avez besoin** pour gérer les problèmes importants. Nous allons découvrir de nouveaux outils comme **l’analyse de domaine** et **les diagrammes de cas d’utilisation**, mais même ces nouveaux outils s’inspirent de ce que vous connaissez déjà, comme écouter le client et comprendre ce que vous allez construire avant de commencer à écrire du code. Préparez-vous... vous allez commencer à jouer à l’architecte.

Résoudre de gros problèmes	280
Tout est dans la façon de regarder le gros problème	281
Les exigences et les cas d’utilisaiton sont un bon point de départ...	286
Conformité et variabilité	287
Définir les caractéristiques	290
La différence entre caractéristiques et exigences	292
Les cas d’utilisation ne vous aident pas toujours à avoir une vue d’ensemble	294
Diagrammes de cas d’utilisation	296
L’acteur caché	301
Les acteurs sont aussi des êtres humains (hum, pas toujours)	302
Faisons une petite analyse de domaine	307
Diviser pour régner	309
N’oubliez pas qui est votre vrai client	313
Qu’est-ce qu’un design pattern ?	315
Le pouvoir de l’ACOO (et d’un peu de bon sens)	318
Boîte à outils ACOO	320



architecture

7 Mettre de l'ordre dans le chaos

Il faut commencer par quelque chose, mais autant choisir la *bonne* ! Vous savez comment fragmenter votre application en plusieurs petits problèmes, mais maintenant vous avez **BEAUCOUP** de petits problèmes. Dans ce chapitre, nous allons vous aider à trouver **par où commencer** et nous assurer que vous ne perdez pas votre temps à travailler sur les mauvais éléments. Il est temps de rassembler toutes ces **petits morceaux** éparpillés sur votre plan de travail et de trouver un moyen de les transformer en une **application bien ordonnée et bien conçue**. En chemin vous découvrirez les **3 questions de l'architecture** mais aussi que le **risque** est bien plus qu'un mot écrit en grosses lettres sur votre contrat d'assurance.

Baromètre au risque

Aucune chance d'être dans les temps.

Une chance sur cent que cela fonctionne.

Très peu de choses peuvent mal tourner.

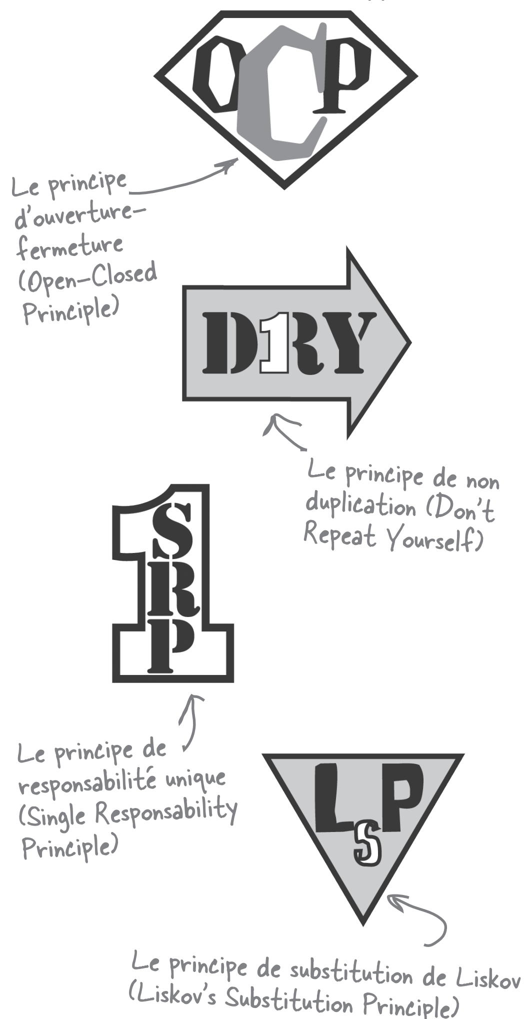
Aussi proche de l'infaillibilité que peut l'être un logiciel !

Vous vous sentez un peu dépassé ?	324
Nous avons besoin d'une architecture	326
Commençons par la fonctionnalité	329
Qu'est-ce qui est significatif dans l'architecture ?	331
Les trois questions de l'architecture	332
Réduire les risques	338
Les scénarios aident à réduire les risques	341
Concentrez-vous sur une caractéristique à la fois	349
L'architecture est la structure de votre conception	351
La conformité revisitée	355
L'analyse de conformité : la route vers un logiciel souple	361
Qu'est-ce que cela veut dire ? Demandez au client.	366
Réduire le risque vous aide à écrire un bon logiciel	371
Points d'impact	372

principes de conception

8 On accorde trop d'importance à l'originalité

L'imitation est la plus intelligente des flatteries. Il n'y a rien de plus satisfaisant que de trouver une solution nouvelle et originale à un problème qui vous a empoisonné pendant des jours. Jusqu'à ce que vous vous rendiez compte que quelqu'un d'autre **avait résolu le même problème** bien avant vous et bien mieux que vous ! Dans ce chapitre, nous allons étudier quelques **principes de conception** qui ont été établis au fil des années et voir comment ils peuvent améliorer vos compétences de programmeur. Laissez de côté l'idée de « travailler à votre façon ». Ce chapitre va vous apprendre à **travailler de la façon la plus rapide et la plus intelligente**.



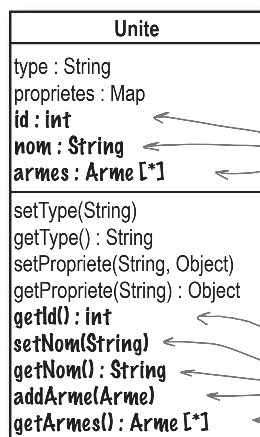
Rassemblement de principes de conception	376
Le principe d'ouverture-fermeture	377
Le principe d'ouverture-fermeture, étape par étape	379
Le principe de non duplication	382
La non duplication : une exigence à un endroit	384
Le principe de responsabilité unique	390
Repérer les responsabilités multiples	392
Des responsabilités multiples à une responsabilité unique	395
Le principe de substitution de Liskov	400
Mauvaise utilisation des sous-classes : une étude de cas des mauvaises utilisations de l'héritage	401
Le principe de substitution de Liskov révèle les problèmes cachés de votre structure d'héritage	402
Les sous-types doivent pouvoir être substitués à leur type de base	403
Enfreindre le principe de substitution de Liskov sème la confusion dans le code	404
Déléguez la fonctionnalité à une autre classe	406
Utilisez la composition pour assembler des comportements d'autres classes	408
L'agrégation : la composition sans la fin brutale	412
L'agrégation ou la composition	413
L'héritage n'est qu'une option	414
Points d'impact	417
Boîte à outils ACOO	418

itérer et tester

9 Le logiciel reste destiné au client

Il est temps de montrer au client à quel point il compte pour vous. Des chefs envahissants ? Des clients inquiets ? Des actionnaires qui demandent sans arrêt : « Est-ce que ce sera prêt à temps ? ». Peu importe la quantité de code bien conçu que vous écrivez, vos clients ne s'y intéresseront pas. Vous devez **leur montrer quelque chose qui fonctionne**. Maintenant que votre boîte à outils de programmation OO est bien remplie, il est temps d'apprendre à **prouver au client** que votre logiciel fonctionne. Dans ce chapitre, nous allons découvrir deux façons de **plonger au cœur** des fonctionnalités de votre logiciel et à donner au client ce sentiment de satisfaction intérieure qui lui fera dire : « ***Oui, vous êtes vraiment le développeur qu'il me faut !*** »

Votre boîte à outils se remplit 424
Vous écrivez un bon logiciel par itération 426
Itérer en profondeur : deux choix fondamentaux 427
Le développement orienté fonctionnalités 428
Le développement orienté cas d'utilisation 429
Deux approches du développement 430
Analyse d'une caractéristique 434
Écrire des scénarios de test 437
Le développement orienté tests 440
Analyse de conformité (revisitée) 442
Priorité à la conformité 446
Priorité à l'encapsulation 448
Reliez vos tests à votre conception 452
Les cas de test disséqués... 454
Faites vos preuves pour le client 460
Nous avons programmé par contrat jusqu'ici 462
Programmer par contrat est une question de confiance 463
Programmation défensive 464
Divisez vos applis en plus petites fonctionnalités 473
Points d'impact 475
Boîte à outils ACOO 478



Toutes les propriétés communes à toutes les unités sont représentées par des variables en dehors de la Map de propriétés.

Samuel a pensé qu'il serait initialisé dans le constructeur d'Unite, il n'a donc pas besoin d'une méthode setId().

Chaque nouvelle propriété a son propre ensemble de méthodes.