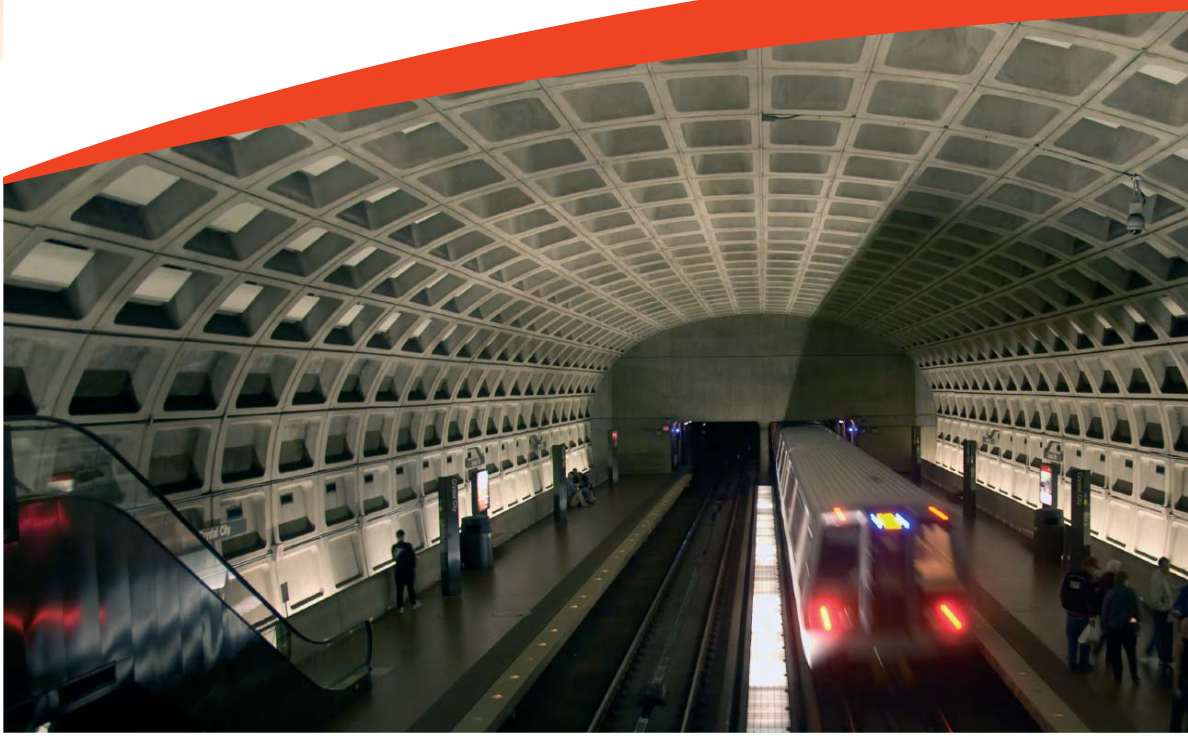
Référence

Les design patterns en Ruby

Russ Olsen



Réseaux et télécom

Programmation

Génie logiciel

Sécurité

Système d'exploitation

Version française par Laurent Julliard, Mikhail Kachakhidze et Richard Piacentini

PEARSON

Les design patterns en Ruby

Russ Olsen

PEARSON
Education

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

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

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

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

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

Mise en pages : TyPAO

ISBN : 978-2-7440-4018-4

Titre original :
Design Patterns in Ruby

Traduit de l'américain par Laurent Julliard,
Mikhail Kachakhidze et Richard Piacentini

ISBN original : 978-0-321-49045-2

À Karen, qui a permis tout ceci,
et à Jackson, qui y donne un sens.

Table des matières

Préface à l'édition française XIII

Préface XV

Avant-propos XVII
- À qui est destiné ce livre ? XIX
- Comment ce livre est-il organisé ? XIX
- Avertissement XIX
- Le style de code utilisé dans ce livre XX

À propos de l'auteur XXIII

Partie I

Patterns et Ruby

Chapitre 1. Améliorer vos programmes avec les patterns 3
- The Gang of Four (la bande des quatre) 4
- Patterns des Patterns 5
  - Séparer ce qui change de ce qui reste identique 5
  - Programmer par interface et non par implémentation 6
  - Préférer la composition à l'héritage 7
  - Déléguer, déléguer, déléguer 11
  - Vous n'aurez pas besoin de ça 12
- Quatorze sur vingt-trois 14
- Patterns en Ruby ? 16

Chapitre 2. Démarrer avec Ruby 19
- Ruby interactif 20
- Afficher Hello World 20
- Variables 22
- Fixnums et Bignums 24
- Nombres à virgule flottante 25

Il n'y a pas de types primitifs ici 26
Mais, parfois, il n'y a pas d'objet 27
Vérité, mensonges et nil 27
Décisions, décisions 29
Boucles 30
Plus de détails sur les chaînes de caractères 32
Symboles 34
Tableaux 35
Tableaux associatifs 36
Expressions régulières 37
Votre propre classe 38
Accès aux variables d'instance 40
Un objet demande : qui suis-je ? 42
Héritage, classes filles et classes mères 42
Options pour les listes d'arguments 43
Modules 45
Exceptions 47
Threads 48
Gérer des fichiers de code source séparés 49
En conclusion 50

Partie II

Patterns en Ruby

Chapitre 3. Varier un algorithme avec le pattern Template Method 53
Faire face aux défis de la vie 54
Séparer les choses qui restent identiques 55
Découvrir le pattern Template Method 58
Méthodes d'accrochage 59
Mais où sont passées toutes les déclarations ? 61
Types, sécurité et flexibilité 61
Les tests unitaires ne sont pas facultatifs 64
User et abuser du pattern Template Method 65
Les Templates dans le monde réel 67
En conclusion 67

Chapitre 4. Remplacer un algorithme avec le pattern Strategy 69
Déléguer, déléguer et encore déléguer 69
Partager les données entre l'objet contexte et l'objet stratégie 72

Typage à la canard une fois de plus 73
Procs et blocs 75
Stratégies rapides et pas très propres 78
User et abuser du pattern Strategy 80
Le pattern Strategy dans le monde réel 80
En conclusion 82

Chapitre 5. Rester informé avec le pattern Observer 83
Rester informé 83
Une meilleure façon de rester informé 85
Factoriser le code de gestion du sujet 88
Des blocs de code comme observateurs 91
Variantes du pattern Observer 92
User et abuser du pattern Observer 92
Le pattern Observer dans le monde réel 94
En conclusion 95

Chapitre 6. Assembler le tout à partir des composants avec Composite 97
Le tout et ses parties 97
Créer des composites 100
Raffiner le pattern Composite avec des opérateurs 103
Un tableau comme composite ? 104
Une différence embarrassante 105
Pointeurs dans les deux sens 106
User et abuser du pattern Composite 107
Les composites dans le monde réel 108
En conclusion 109

Chapitre 7. Accéder à une collection avec l'Itérateur 111
Itérateurs externes 111
Itérateurs internes 113
Itérateurs internes versus itérateurs externes 114
L'incomparable Enumerable 116
User et abuser du pattern Itérateur 117
Les itérateurs dans le monde réel 119
En conclusion 122

Chapitre 8. Effectuer des actions avec Command 123
L'explosion de sous-classes 124
Un moyen plus simple 125
Des blocs de code comme commandes 126

Les commandes d'enregistrement 127
Annuler des actions avec Command 130
Créer des files de commandes 132
User et abuser du pattern Command 133
Le pattern Command dans le monde réel 134
Migration ActiveRecord 134
Madeleine 135
En conclusion 138

Chapitre 9. Combler le fossé avec l'Adapter 139
Adaptateurs logiciels 140
Les interfaces presque parfaites 142
Une alternative adaptative ? 144
Modifier une instance unique 145
Adapter ou modifier ? 147
User et abuser du pattern Adapter 147
Le pattern Adapter dans le monde réel 148
En conclusion 149

Chapitre 10 . Créer un intermédiaire pour votre objet avec Proxy 151
Les proxies à la rescousse 152
Un proxy de contrôle d'accès 153
Des proxies distants 155
Des proxies virtuels à vous rendre paresseux 156
Éliminer les méthodes ennuyeuses des proxies 157
Les méthodes et le transfert des messages 158
La méthode method_missing 159
Envoi des messages 160
Proxies sans peine 160
User et abuser du pattern Proxy 163
Proxies dans le monde réel 164
En conclusion 165

Chapitre 11. Améliorer vos objets avec Decorator 167
Décorateurs : un remède contre le code laid 167
La décoration formelle 172
Diminuer l'effort de délégation 173
Les alternatives dynamiques au pattern Decorator 174
Les méthodes d'encapsulation 174
Décorer à l'aide de modules 175
User et abuser du pattern Decorator 176

Les décorateurs dans le monde réel 177
En conclusion 178

Chapitre 12. Créer un objet unique avec Singleton 179
Objet unique, accès global 179
Variables et méthodes de classe 180
Variables de classe 180
Méthodes de classe 181
Première tentative de création d'un singleton Ruby 182
Gestion de l'instance unique 183
S'assurer de l'unicité 184
Le module Singleton 184
Singletons à instanciation tardive ou immédiate 185
Alternatives au singleton classique 185
Variables globales en tant que singletons 185
Des classes en tant que singletons 186
Des modules en tant que singletons 188
Ceinture de sécurité ou carcan ? 188
User et abuser du pattern Singleton 190
Ce sont simplement des variables globales, n'est-ce pas ? 190
Vous en avez combien, des singletons ? 190
Singletons pour les intimes 191
Un remède contre les maux liés aux tests 193
Les singletons dans le monde réel 193
En conclusion 194

Chapitre 13. Choisir la bonne classe avec Factory 195
Une autre sorte de typage à la canard 196
Le retour du pattern Template Method 198
Des méthodes factory avec des paramètres 200
Les classes sont aussi des objets 202
Mauvaise nouvelle : votre programme a du succès 203
Création de lots d'objets 204
Des classes sont des objets (encore) 207
Profiter du nommage 208
User et abuser des patterns Factory 209
Les factories dans le monde réel 209
En conclusion 211

Chapitre 14. Simplifier la création d'objets avec Builder 213
Construire des ordinateurs 214

Des objets builder polymorphes 216
Les builders peuvent garantir la validité des objets 219
Réutilisation de builders 219
Améliorer des objets builder avec des méthodes magiques 220
User et abuser du pattern Builder 221
Des objets builder dans le monde réel 222
En conclusion 223

Chapitre 15. Assembler votre système avec Interpreter 225
Langage adapté à la tâche 226
Développer un interpréteur 226
Un interpréteur pour trouver des fichiers 229
Retrouver tous les fichiers 229
Rechercher des fichiers par nom 230
Des grands fichiers et des fichiers ouverts en écriture 231
Des recherches plus complexes à l'aide des instructions Not, And et Or 232
Créer un AST 234
Un analyseur syntaxique simple 234
Et un interpréteur sans analyseur ? 236
Déléguer l'analyse à XML ou à YAML ? 237
Racc pour des analyseurs plus complexes 238
Déléguer l'analyse à Ruby ? 238
User et abuser du pattern Interpreter 238
Des interpréteurs dans le monde réel 239
En conclusion 240

Partie III

Les patterns Ruby

Chapitre 16. Ouvrir votre système avec des langages spécifiques d'un domaine 245
Langages spécifiques d'un domaine 245
Un DSL pour des sauvegardes de fichiers 246
C'est un fichier de données, non, c'est un programme ! 247
Développer PackRat 248
Assembler notre DSL 250
Récolter les bénéfices de PackRat 251
Améliorer PackRat 252
User et abuser des DSL internes 254
Les DSL internes dans le monde réel 254
En conclusion 256

Chapitre 17. Créer des objets personnalisés par méta-programmation 257
Des objets sur mesure, méthode par méthode 258
Des objets sur mesure, module par module 259
Ajouter de nouvelles méthodes 261
L'objet vu de l'intérieur 264
User et abuser de la méta-programmation 265
La méta-programmation dans le monde réel 266
En conclusion 269

Chapitre 18. Convention plutôt que configuration 271
Une bonne interface utilisateur... pour les développeurs 273
Anticiper les besoins 273
Ne le dire qu'une seule fois 274
Fournir un modèle 274
Une passerelle de messages 275
Sélectionner un adaptateur 277
Charger des classes 278
Ajouter un niveau de sécurité 281
Aider un utilisateur dans ses premiers pas 283
Récolter les bénéfices de la passerelle de messages 284
User et abuser du pattern Convention plutôt que configuration 284
Convention plutôt que configuration dans le monde réel 285
En conclusion 286

Conclusion 287

Annexes

Annexe A. Installer Ruby 291
Installer Ruby sous Microsoft Windows 291
Installer Ruby sous Linux ou un autre système de type UNIX 291
Mac OS X 292

Annexe B. Aller plus loin 293
Design patterns 293
Ruby 294
Expressions régulières 295
Blogs et sites Web 296

À propos des traducteurs 297

Index 299

Préface à l'édition française

Design Patterns In Ruby started out as a 900 word blog article that I wrote in one afternoon. I certainly never dreamed that those 17 or so paragraphs would lead to a book in English, let alone to a French edition. Perhaps this is not so surprising, because that is what Ruby is like: Ruby changes the odds, it makes the difficult easy and many impossible things possible. But the language is only half the story: Every programming language needs an enthusiastic user community to be successful. Here too Ruby has been specially blessed. Certainly I am grateful to Richard Piacentini, Laurent Julliard and Mikhail Kachakhidze for making this French edition possible.

Russ Olsen
Virginia, April 2008

C'est avec un plaisir non dissimulé que nous livrons aux lecteurs francophones cette traduction de l'ouvrage de Russ Olsen sur les patrons de conception (*design patterns*) en Ruby. Et ce pour plusieurs raisons.

Tout d'abord parce que nous disposons désormais d'un ouvrage supplémentaire en français qui met en avant le langage Ruby. En effet, si les publications sur le framework web *Ruby on Rails* sont aujourd'hui légion (plus de 20 titres en français à ce jour !), il n'en va pas de même pour le langage Ruby qui ne compte que quelques titres.

Ensuite parce que l'adaptation à Ruby d'un des ouvrages les plus célèbres de l'histoire de l'informatique, *Design Patterns* de Erich Gamma, Richard Helm, Ralph Johnson et John M. Vlissides (souvent appelé "le GoF", abréviation de *The Gang of Four* ou "la bande des quatre", en référence aux quatre auteurs de l'ouvrage) est une indéniable marque de maturité du langage Ruby lui-même, mais aussi de sa communauté. Les centaines de milliers de développeurs qui ont découvert Rails au cours des deux dernières années ont aussi pris conscience que derrière Rails se cache Ruby, un langage de programmation totalement orienté objet, incroyablement agile et expressif qu'on peut utiliser dans de très nombreux domaines (algorithmie, système, réseau, modélisation, etc.).

Les patrons de conception sont d'ailleurs un thème rêvé pour mettre en lumière les qualités de ce langage créé en 1995 par un universitaire japonais, Yukihiro Matsumoto ("Matz" pour les intimes...). Vous pourrez notamment constater à quel point l'expressivité et les capacités dynamiques du langage Ruby permettent de s'affranchir des lourdeurs et du code fastidieux souvent nécessaire à la mise en œuvre de ces mêmes patrons de conception dans d'autre langages comme C++ ou Java. Ces qualités ont aussi inspiré à la communauté Ruby la création de nouveaux patrons de conception comme les DSL (langages spécifiques d'un domaine) ou "Convention plutôt que Configuration", que vous découvrirez à la fin de l'ouvrage.

Avant de vous laisser à la lecture de ce livre qui est déjà une référence outre-Atlantique, nous aimerions vous mettre en garde et vous proposer un sujet de réflexion. L'avertissement porte sur le caractère terriblement "addictif" du langage Ruby : ceux d'entre vous qui s'apprêtent à côtoyer Ruby pour la première fois risquent fort d'en ressentir les effets secondaires, c'est-à-dire une certaine aversion pour les langages à typage statique utilisés aujourd'hui dans l'industrie, comme C++, C# ou Java.

Pour vous aider à surmonter cette impression de clivage profond, nous terminerons en vous livrant quelques éléments de réflexion. Sun et Microsoft ont tout deux lancé le portage du langage Ruby sur leur machine virtuelle (respectivement Jruby et Ruby-DLR) et Apple livre désormais Ruby et Rails en standard avec ces kits de développement. Tous ont compris que dans certains domaines les atouts du langage Ruby permettent des gains de productivité considérables sans pour autant hypothéquer ni l'existant ni la qualité des logiciels produits.

Les langages dynamiques orientés objet, apparus dans les années 1970 avec Smalltalk, sont à l'origine de la plupart des concepts de programmation utilisés aujourd'hui. Longtemps éclipsés par les langages "poids lourd" (dans tous les sens du terme !) de l'industrie, les voici qui reviennent en force sur le devant de la scène et Ruby en est assurément l'un des plus dignes représentants.

Bonne lecture !

Laurent Julliard – Richard Piacentini

Préface

Design patterns. Catalogue de modèles de conceptions réutilisables, connu sous le nom affectueux de "livre du Gang of Four" (ou GoF), est le premier ouvrage de référence publié sur ce sujet, devenu depuis très populaire. Avec plus d'un demi-million d'exemplaires vendus, cet ouvrage a sans doute influencé la façon de penser et de coder de millions de programmeurs dans le monde entier. Je me rappelle distinctement le jour où j'ai acheté mon premier exemplaire de ce livre à la fin des années 1990. En partie à cause des recommandations enthousiastes de mes amis, je l'ai considéré comme une étape incontournable vers ma maturité en tant que programmeur. J'ai avalé le livre en quelques jours en essayant d'inventer des applications pratiques pour chacun des patterns.

Il est généralement reconnu que la caractéristique la plus utile des patterns réside dans le vocabulaire qu'ils proposent. Ce vocabulaire permet aux programmeurs d'exprimer des modèles de conception dans les conversations qu'ils peuvent avoir entre eux en cours de développement. C'est particulièrement vrai pour la programmation en paire (*pair programming*), la pierre angulaire de la programmation agile de type Extreme Programming, ainsi que pour d'autres techniques agiles, où la conception est une activité quotidienne et collective. Il est fantastiquement pratique de pouvoir dire à votre collègue "Je pense qu'ici nous avons besoin d'une stratégie" ou "Ajoutons cette fonctionnalité sous la forme d'un observateur".

Dans certaines entreprises, la connaissance de design patterns est même devenue un moyen simple pour filtrer des candidats :

"Quel est votre pattern préféré ?"
"Hum... Factory ?"
"Merci d'être venu, au revoir."

Il est vrai que la notion de pattern préféré est assez étrange. Votre pattern préféré doit être par définition celui qui est le plus adapté aux circonstances. Une des erreurs classiques faites par des programmeurs inexpérimentés qui commencent à apprendre les patterns est d'implémenter un pattern comme une fin en soi et non pas comme un outil. Pourquoi implémenter des patterns dans le but de s'amuser ?

Dans le monde des langages à typage statique, l'implémentation de design patterns présente un certain nombre de défis techniques. Dans le meilleur des cas, vous allez utiliser des techniques de ninja pour démontrer toute votre habileté au codage. Dans le pire des scénarii, vous vous retrouverez avec un tas de code générique totalement répugnant. Pour des allumés de la programmation comme moi, cela suffit à rendre le sujet sur les design patterns particulièrement amusant.

Est-ce que les design patterns du GoF sont difficiles à implémenter en Ruby ? Pas vraiment. Tout d'abord, l'absence de typage statique réduit le coût de vos programmes en terme de lignes de code. La bibliothèque standard de Ruby permet d'inclure les patterns les plus fréquents en une ligne, alors que les autres sont essentiellement incorporés dans le langage même. Par exemple, selon le GoF un objet Command encapsule du code qui sait effectuer une tâche ou exécuter un fragment de code à un moment donné. Cette description correspond également à un bloc de code – un Proc – en Ruby.

Russ a travaillé avec Ruby depuis 2002 et il sait que la majorité des développeurs Ruby expérimentés ont déjà une bonne maîtrise des design patterns et de leurs applications. D'après moi, son défi principal consistait à écrire un livre qui soit à la fois pertinent pour des programmeurs Ruby professionnels mais qui puisse aussi profiter aux débutants. Je pense qu'il a réussi et je suis convaincu que vous serez d'accord avec moi sur ce point. Prenons notre exemple d'un objet Command : dans sa forme simple, il peut être facilement implémenté avec un bloc Ruby, mais il suffit d'y ajouter de l'information sur son état et un peu de code métier et l'implémentation devient tout de suite plus complexe. Russ nous fournit des conseils avertis spécifiques à Ruby et prêts à être appliqués.

Ce livre présente aussi l'avantage d'inclure de nouveaux design patterns spécifiques à Ruby. Russ a identifié et expliqué en détail plusieurs modèles de conception dont un de mes préférés : Internal Domain Specific Languages (les langages internes spécifiques d'un domaine). Je crois que sa vision de ce pattern comme l'évolution du pattern Interpreter est une analyse de référence significative et sans précédent.

Enfin, je pense que ce livre sera extrêmement bénéfique à ceux qui débutent leur carrière dans le monde Ruby ou qui migrent d'autres langages, comme PHP, qui ne mettent pas autant l'accent sur les pratiques de conception orientée objet. En décrivant les design patterns, Russ a illustré des solutions essentielles aux problèmes que l'on rencontre quotidiennement dans le développement de programmes d'envergure en Ruby : ce sont des conseils d'une valeur inestimable pour un débutant. Je suis sûr que ce livre sera le cadeau que j'offrirai en priorité à mes amis et collègues débutants.

Obie Fernandez, éditeur de la série Professional Ruby

Avant-propos

Un ancien collègue disait que les gros livres sur les design patterns témoignent de l'inadéquation d'un langage de programmation. Il entendait par là que les patterns sont des idiomes courants dans le code et qu'un bon langage de programmation doit rendre leur implémentation extrêmement simple. Un langage idéal intégrerait les design patterns jusqu'au point de les rendre quasiment transparents. Pour vous donner un exemple extrême, à la fin des années 1980 j'ai travaillé sur un projet où l'on produisait du code C orienté objet. Oui, C et non pas C++. Voici comment nous avons réussi cet exploit. Chaque objet (en réalité une structure en C) pointait vers un tableau de pointeurs de fonction. Pour utiliser l'objet nous retrouvions le tableau correspondant et nous appelions ses fonctions, en simulant ainsi des appels de méthodes. C'était étrange et pas très propre, mais cela fonctionnait.

Si nous y avions pensé, nous aurions pu appeler cette technique "le pattern orienté-objet". Évidemment, avec l'arrivée de C++, puis de Java, notre modèle s'est incorporé si profondément au langage qu'il est devenu invisible. Aujourd'hui, l'orientation objet n'est plus considérée comme un pattern – c'est trop simple.

Pourtant, beaucoup de choses demeurent encore complexes.

Design patterns. Catalogue de modèles de conceptions réutilisables, écrit par Gamma, Helm, Johnson et Vlissides, qui a acquis une notoriété bien méritée, fait partie d'un programme de lecture obligatoire pour chaque ingénieur logiciel. Or l'implémentation des modèles décrits dans cet ouvrage avec les langages répandus (Java, C++ et peut-être C#) ressemble beaucoup au système "à l'ancienne" que j'ai conçu dans les années 1980. Trop pénible. Trop verbeux. Trop enclin aux bugs.

Le langage de programmation Ruby se rapproche davantage de l'idéal de mon vieil ami – il facilite si bien l'implémentation des patterns que la plupart du temps ceux-ci se fondent dans l'arrière-plan.

Cette facilité est due à plusieurs facteurs :

- Ruby est un langage dynamiquement typé. En supprimant le typage statique, Ruby réduit de façon significative le surcoût de code nécessaire à la construction de la plupart des programmes, y compris ceux qui implémentent des patterns.
- Ruby a des fermetures lexicales. Il permet de passer des fragments de code avec leur environnement sans recourir à la construction de classes et d'objets inutiles.
- Les classes Ruby sont de vrais objets. Le fait qu'une classe soit un objet comme un autre nous permet de manipuler une classe Ruby au moment de l'exécution. On peut créer de nouvelles classes ou modifier des classes existantes en ajoutant ou en supprimant ses méthodes. On peut même cloner une classe et modifier la copie sans toucher à l'original.
- Ruby présente un modèle élégant de la réutilisation de code. En plus de l'héritage classique, Ruby permet de définir des mixins qui fournissent un moyen simple mais flexible d'écrire du code partageable entre plusieurs classes.

Tout cela rend le code Ruby très compact. En Ruby, tout comme en Java et C++, on peut implémenter des idées très sophistiquées, mais Ruby propose des moyens beaucoup plus efficaces de cacher les détails de leur implémentation.

Comme vous le verrez par la suite, de nombreux "design patterns" qui nécessitent d'interminables lignes de code générique dans les langages statiques traditionnels ne requièrent qu'une ou deux lignes en Ruby. Vous pouvez transformer une classe en un singleton avec la simple commande *include Singleton*. Vous pouvez déléguer aussi facilement qu'hériter. Ruby vous donne les outils pour exprimer davantage de choses intéressantes à chaque ligne, ce qui réduit votre base de code.

Il s'agit non seulement d'éviter de taper sur le clavier mais aussi d'appliquer le principe DRY (*Don't Repeat Yourself*).

Je doute que quelqu'un dans le monde d'aujourd'hui regrette mon vieux pattern orienté-objet en C. Il fonctionnait, mais m'a demandé beaucoup d'efforts. De même, les implémentations traditionnelles des nombreux "design patterns" fonctionnent, mais vous demandent beaucoup d'efforts. Ruby représente un vrai pas en avant car il suffit de faire le travail une fois et de le dissocier de votre code principal. En résumé, Ruby permet de se concentrer sur des solutions à des problèmes concrets, au lieu de la tuyauterie. Je souhaite que ce livre vous montre comment y parvenir.

À qui est destiné ce livre ?

Ce livre s'adresse aux programmeurs qui souhaitent apprendre à développer des applications en Ruby. Les concepts de base de la programmation orientée objet doivent être connus, mais vous n'aurez besoin d'aucune connaissance particulière en matière de design patterns. Vous pourrez les apprendre en lisant ce livre.

Vous n'aurez pas besoin non plus d'une maîtrise approfondie de Ruby pour tirer pleinement parti de ce livre. Une introduction rapide au langage vous est proposée au Chapitre 2, les autres points spécifiques à Ruby étant expliqués au fil de l'ouvrage.

Comment ce livre est-il organisé ?

Le présent ouvrage est divisé en trois parties. La Partie 1 est constituée de deux chapitres d'introduction : le premier passe en revue l'historique et les raisons qui ont présidé à la naissance des design patterns et le second vous propose un tour d'horizon du langage Ruby suffisamment étoffé pour que vous "deveniez dangereux".

La Partie 2, qui représente la majeure partie de ce livre, examine du point de vue Ruby un certain nombre de patterns du Gang of Four. Quels sont les problèmes que résout un pattern ? À quoi ressemblent l'implémentation traditionnelle – celle fournie par le Gang of Four – et celle en Ruby ? Le pattern est-il justifié en Ruby ? Existe-t-il des alternatives en Ruby pour faciliter la solution à ce problème ? Autant de questions auxquelles nous apportons des réponses dans cette seconde partie.

La Partie 3 couvre trois patterns qui sont apparus avec l'usage avancé de Ruby.

Avertissement

Je ne peux pas signer de mon nom un livre sur les design patterns sans répéter le mantra que je murmure depuis des années : les design patterns sont des solutions "précuites" aux problèmes courants de programmation. Idéalement, lorsque vous rencontrez le bon problème il suffit d'appliquer le bon design pattern et vous avez la solution. C'est cette première partie de la phrase – attendre de rencontrer le "bon problème" – qui semble poser problème à certains ingénieurs. On ne peut pas considérer qu'un pattern est appliqué correctement si le problème que ce pattern est censé résoudre n'existe pas. L'usage imprudent de tous les patterns pour régler des problèmes qui n'en sont pas est à l'origine de la mauvaise réputation que se sont taillés les design patterns dans certains milieux. Je me permets d'affirmer qu'en Ruby il est plus facile d'écrire un adaptateur qui utilise une factory pour obtenir un proxy d'un builder, qui crée à son tour une commande pour coordonner l'opération en vue d'additionner deux plus deux.

C'est vrai, Ruby rend le processus effectivement plus facile, mais même en Ruby cela n'a aucun sens.

On ne peut non plus voir la construction de programmes comme un processus de recombinaison de patterns existants. Tout programme intéressant a des sections uniques : du code qui est parfait pour un problème donné et aucun autre. Les design patterns sont là pour vous aider à reconnaître et à résoudre des problèmes de conception fréquents et répétitifs. L'avantage des design patterns, c'est que vous pouvez rapidement évacuer des problèmes que quelqu'un a déjà résolus pour passer aux choses véritablement difficiles, à savoir le code spécifique à votre situation. Les patterns ne sont pas une potion magique pour régler tous vos soucis de conception. Ils ne sont qu'une technique – une technique très utile – que vous pouvez utiliser en développant des programmes.

Le style de code utilisé dans ce livre

Si la programmation en Ruby est si agréable, c'est que le langage tente de s'effacer. S'il existe plusieurs moyens sensés d'exprimer quelque chose, Ruby, en général, les propose tous :

```
# Une façon de l'exprimer
if (divisor == 0)
  puts 'Division by zero'
end
# Encore une
puts 'Division by zero' if (divisor == 0)
# Et une troisième
(divisor == 0) && puts('Division by zero')
```

Ruby n'essaie pas d'insister sur l'utilisation méticuleuse de la syntaxe. Lorsque le sens de l'instruction est clair, Ruby permet d'omettre des éléments syntaxiques. Par exemple, d'habitude vous pouvez omettre des parenthèses dans les listes d'arguments quand vous appelez une méthode :

```
puts('A fine way to call puts')
puts 'Another fine way to call puts'
```

Vous pouvez même omettre des parenthèses lorsque vous définissez la liste d'arguments d'une méthode ou d'une expression conditionnelle :

```
def method_with_3_args a, b, c
  puts "Method 1 called with #{a} #{b} #{c}"
  if a == 0
    puts 'a is zero'
  end
end
```

Le problème de ces raccourcis vient du fait que leur usage excessif a tendance à embrouiller les débutants. Une majorité des programmeurs qui débutent en Ruby seraient plus rassurés par

```
if file.eof?
  puts( 'Reached end of file' )
end
```

ou même

```
puts 'Reached end of file' if file.eof?
```

que par

```
file.eof? && puts('Reached end of file')
```

Puisque ce livre se concentre plus sur la puissance et l'élégance inhérentes à Ruby que sur la syntaxe, j'ai essayé de faire en sorte que mes exemples ressemblent à du code Ruby réel tout en restant compréhensibles par des débutants. En pratique, cela signifie que je profite des raccourcis les plus évidents mais que j'évite volontairement des astuces plus radicales. Cela ne veut pas dire que je ne sois pas au courant ou que je ne soutienne pas l'utilisation de la "sténographie" Ruby. Je suis simplement plus concentré sur la nécessité de faire passer le message de l'élégance conceptuelle de ce langage auprès des lecteurs débutants. Vous aurez beaucoup d'occasions d'apprendre des raccourcis lorsque vous serez tombé fou amoureux de ce langage.

À propos de l’auteur

Russ Olsen est ingénieur logiciel avec à son actif plus de vingt-cinq ans d’expérience. Russ a développé des logiciels dans des domaines aussi variés que la conception ou la production assistée par ordinateur, les systèmes d’information géographique, la gestion de documents et l’intégration des systèmes d’information en entreprise. Il travaille actuellement sur des solutions de sécurité et de découverte automatique de services SOA pour de grandes entreprises.

Russ a contribué à Ruby depuis 2002. Il est l’auteur de ClanRuby, une tentative à l’époque d’ajouter des fonctionnalités multimédias à Ruby. Aujourd’hui, il participe à de nombreux projets open source y compris UDDI4R.

Des articles techniques de Russ sont parus dans *Javalobby, On Java* chez O’Reilly ainsi que sur le site de *Java Developer’s Journal*.

Russ est diplômé de Temple University et habite en banlieue de Washington avec sa famille, deux tortues et un nombre indéterminé de guppy.

Russ est joignable par courrier électronique à russ@russolsen.com.

Partie I

Patterns et Ruby

1

Améliorer vos programmes avec les patterns

C’est drôle, mais les "design patterns" me rappellent toujours une certaine épicerie. Encore lycéen, j’avais décroché un premier boulot à temps partiel. Pendant quelques heures les jours de la semaine et le samedi toute la journée, je donnais un coup de main dans un petit commerce local. Toutes les tâches peu qualifiées telles que l’approvisionnement des étagères ou encore le balayage du plancher faisaient partie de mes responsabilités. Au début, la vie de cette petite épicerie me paraissait un étrange mélange d’images (je n’ai jamais apprécié l’aspect d’un foie cru), de sons (mon patron avait été instructeur dans le corps des marines américains et il savait impressionner par sa voix) et d’odeurs (je vous passe les détails).

Pourtant, plus je travaillais chez Conrad Market, plus ces événements isolés se regroupaient pour former des procédures compréhensibles. Le matin, il fallait ouvrir la porte d’entrée, éteindre l’alarme et afficher le panneau "Oui ! Nous sommes ouverts". Le soir, le processus était inversé. Entre les deux je m’occupais d’un million de choses : approvisionner les étagères, aider les clients à trouver le ketchup – tout et n’importe quoi. Au fur et à mesure que je faisais connaissance avec mes homologues dans les autres commerces, je découvrais que leur mode de fonctionnement était quasiment le même.

Cela illustre la façon dont les gens réagissent face aux problèmes et à la complexité de la vie. Nous improvisons et trouvons des solutions rapides à de nouveaux problèmes, mais pour répondre aux problèmes récurrents nous développons des procédures standardisées. Il ne faut pas réinventer la roue, comme dit le proverbe.

The Gang of Four (la bande des quatre)

Réinventer la roue est un réel problème pour des ingénieurs logiciel. Personne n'aime faire et refaire la même chose, mais lorsqu'on conçoit des systèmes il est parfois difficile de se rendre compte que le dispositif rotatoire de réduction de friction à axe central que l'on vient de réaliser est effectivement une roue. La conception de logiciels peut devenir tellement complexe qu'il est facile de ne pas reconnaître les problèmes types qui se présentent à nous de façon répétée et leurs solutions récurrentes.

En 1995, Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides ont décidé de consacrer leur énergie à un travail plus utile que la construction répétée de "roues". Sur la base du travail effectué entre autres par Christopher Alexander et Kent Beck, ils ont publié l'ouvrage *Design patterns. Catalogue de modèles de conceptions réutilisables*. L'ouvrage a été un succès instantané, rendant ses auteurs célèbres (au moins dans le monde du développement logiciel) sous le nom de "The Gang of Four" (le GoF ou la bande des quatre).

Les membres du GoF ont accompli deux tâches. Premièrement, ils ont fait découvrir à un grand nombre d'ingénieurs logiciel la notion des "design patterns" (modèles de conception), un pattern étant une solution "prêt-à-porter" à un problème fréquent de conception. Ils ont écrit : "Nous devons regarder autour de nous et identifier les solutions courantes aux problèmes courants. Nous devons nommer chaque solution et décrire les situations pour lesquelles son utilisation est appropriée, ainsi que les cas où il faut opter pour une approche différente. Cette information sera consignée par écrit afin que la palette des modèles de conception s'étoffe avec le temps."

Deuxièmement, les membres du GoF ont identifié, nommé et décrit vingt-trois patterns initiaux, qu'ils ont considérés comme des modèles clés pour développer des programmes orientés objet propres et bien conçus. Depuis la publication de *Design Patterns*, de nombreux livres ont suivi sur le sujet, décrivant des patterns dans des domaines allant des microcontrôleurs temps réel aux architectures d'entreprise. Pourtant, les vingt-trois patterns recensés par le GoF sont restés au cœur de la conception des logiciels orientés objet en se concentrant sur des questions essentielles : comment les objets dans la plupart des systèmes se réfèrent l'un à l'autre ? Comment doivent-ils être couplés ? Quelles informations sur les autres objets doivent-ils avoir ? Comment remplacer des parties qui sont susceptibles de changer fréquemment ?

Patterns des Patterns

En réponse à ces questions, les auteurs de *Design Patterns* ont formulé plusieurs principes généraux : des méta-design patterns. Pour moi, ces idées se réduisent à quatre points :

- séparer ce qui change de ce qui reste identique ;
- programmer par interface et non par implémentation ;
- préférer la composition à l'héritage ;
- déléguer, déléguer, déléguer.

J'y ajouterai un point qui ne figure pas dans *Design Patterns*, mais qui définit souvent mon approche de la conception de programmes :

- Vous n'aurez pas besoin de ça.

Dans les sections suivantes, nous allons étudier comment chacun de ces principes conditionne la conception de logiciels.

Séparer ce qui change de ce qui reste identique

Le développement de logiciels serait beaucoup plus simple si les choses ne changeaient pas. On pourrait écrire des classes en étant sûr qu'une fois terminées elles continueront à servir à la même tâche. Mais le monde bouge, et dans le monde du développement logiciel c'est encore plus vrai. Les évolutions du matériel informatique, des systèmes d'exploitation, des compilateurs ainsi que les corrections de bugs et les exigences qui varient perpétuellement ont toutes un rôle à jouer. L'objectif clé des ingénieurs logiciel est de développer des systèmes permettant de limiter les dégâts. Dans un système idéal, tout changement serait local : on ne devrait jamais avoir besoin de revérifier la totalité du code si le point A a été modifié, ce qui vous a amené à modifier le point B, qui a déclenché un changement de C, qui a eu une répercussion sur Z. Comment atteindre ou au moins s'approcher du système idéal, dans lequel tout changement serait local ?

Pour atteindre ce but il faut séparer les parties variables de celles qui restent identiques. Si vous pouvez identifier quels aspects de votre système sont susceptibles de changer, vous pourrez les isoler des parties plus stables. Il faudra toujours modifier le code lorsque les besoins évolueront ou lorsqu'on corrigera une anomalie, mais il y a peut-être une chance pour que l'on puisse contenir les modifications dans ces portions de code que nous aurons isolées et protégées afin que le reste demeure inchangé.

Mais comment maintenir cette quarantaine et prévenir la contamination des parties stables par des aspects variables ?

Programmer par interface et non par implémentation

Écrire du code faiblement couplé est toujours un bon début. Si nos classes sont conçues pour une tâche non triviale, elles doivent être au courant l'une de l'autre. Mais que doivent-elles savoir l'une de l'autre exactement ? Le fragment de code Ruby suivant[1] crée une instance de classe `Car` et appelle la méthode d'instance `drive` :

```
my_car = Car.new
my_car.drive(200)
```

Il est clair que ce code est fortement lié à la classe `Car`. Il continuera à fonctionner si l'on a besoin de manipuler un seul véhicule. Si les exigences changent et qu'un autre mode de transport doit être géré (par exemple l'avion), nous nous retrouvons face à un problème. Pour maîtriser deux types de transport on pourrait par exemple se contenter d'écrire l'horreur suivante :

```
# Gérer des voitures et des avions

if is_car
  my_car = Car.new
  my_car.drive(200)
else
  my_plane = AirPlane.new
  my_plane.fly(200)
end
```

Ce code n'est pas seulement sale, mais il est aussi fortement couplé à la fois aux voitures et aux avions. Cette structure pourrait tenir la route jusqu'à l'arrivée d'un bateau, ou d'un train, ou d'un vélo. Une meilleure solution serait de se rappeler les bases de la programmation orientée objet et d'ajouter une dose généreuse de polymorphisme. Si les voitures, les avions et les bateaux implémentent une interface commune, le code peut être amélioré de façon suivante :

```
my_vehicle = get_vehicle
my_vehicle.travel(200)
```

En plus d'être du bon code orienté objet bien lisible, cet exemple démontre le principe de *programmation par interface*.

---

1. Ne vous inquiétez pas, si vous démarrez avec Ruby. Le code utilisé dans ce chapitre est très basique. Le chapitre suivant présente le langage plus en détail. Il ne faut pas s'inquiéter si tout n'est pas complètement clair pour le moment.

Le code initial fonctionnait avec un seul type de véhicule – une voiture –, mais la nouvelle version gère tout objet de type `Vehicle`.

Les développeurs Java et C# suivent parfois ce conseil à la lettre, puisque la définition d'interfaces est un élément de ces langages. Ils extraient soigneusement toute la fonctionnalité principale dans plusieurs interfaces séparées, qui peuvent ensuite être implémentées par des classes concrètes. Généralement, c'est une bonne pratique, mais ce n'est pas vraiment la philosophie sous-jacente au principe de programmation par interface. L'idée consiste à choisir le type le plus générique possible, à ne pas appeler la voiture une voiture s'il est possible de l'appeler un véhicule, peu importe si `Car` et `Vehicle` sont des classes concrètes ou des interfaces abstraites. C'est même mieux si l'on peut choisir un supertype encore plus générique, par exemple un objet mobile. Comme vous le verrez par la suite, le langage Ruby, qui n'inclut pas d'interfaces dans sa syntaxe[1], vous encourage cependant à programmer des interfaces utilisant des supertypes les plus génériques possible.

En écrivant du code avec des types génériques, par exemple en considérant tous les avions, les trains et les voitures comme des véhicules, nous réduisons le couplage de notre code. Au lieu d'avoir quarante-deux classes fortement liées à des voitures, des bateaux et des avions, on finira peut-être avec quarante classes qui ne manipuleront que la notion de véhicule. Il est probable que l'on soit embêté s'il faut ajouter un nouveau type de véhicule correspondant aux deux classes qui restent, mais au moins on aura limité les dégâts. Le fait d'ajouter seulement deux classes indique que nous avons réussi le pari de séparer les parties variables (les deux classes) des parties stables (les quarante autres). Grâce au faible couplage, nous diminuons considérablement le risque que la moindre modification déclenche une réaction en chaîne dévastant la totalité du code.

Néanmoins, programmer des interfaces n'est pas la seule mesure à prendre pour rendre le code résistant aux changements. Il existe aussi la composition.

Préférer la composition à l'héritage

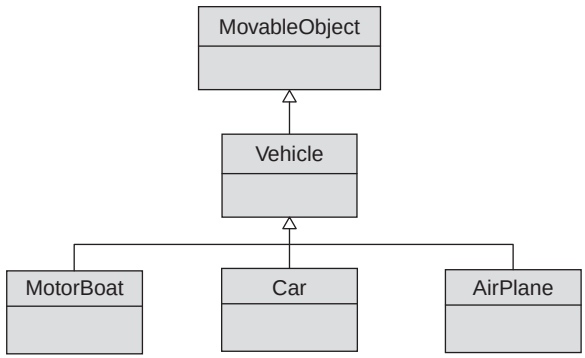
Si votre introduction à la programmation orientée objet a ressemblé à la mienne, vous avez dû passer 10 minutes sur la dissimulation de l'information, 22 minutes sur les questions de visibilité et de portée et le reste du semestre sur l'héritage. Une fois acquises les notions basiques des objets, champs et méthodes, le principal sujet intéressant qui reste est l'héritage, la partie la plus orientée objet de la programmation objet.

1. Le langage Ruby inclut des modules qui ressemblent à des interfaces Java. Vous trouverez plus de détails sur des modules Ruby au Chapitre 2.

L'héritage rend possible l'implémentation sans peine, il suffit de sous-classer la classe `Widget` pour obtenir comme par magie l'accès à toutes ses fonctionnalités.

L'héritage semble être la panacée. Besoin d'implémenter une voiture ? Il suffit de sous-classer `Vehicle`, qui lui-même est de type `MovableObject`, etc. De même, d'autres branches telles que `AirPlane` et `MotorBoat` peuvent apparaître à côté (voir Figure 1.1). À chaque niveau nous profitons des fonctionnalités de la classe mère.

Figure 1.1
Profiter au maximum de l'héritage



Toutefois, l'héritage apporte son lot de défauts. Lorsque vous définissez une sous-classe d'une classe existante, au lieu de créer deux entités séparées vous obtenez deux classes qui sont intimement liées par l'implémentation commune. Par nature, l'héritage a tendance à lier la classe fille et la classe mère. Il y a de fortes chances qu'une modification de comportement de la classe mère affecte le fonctionnement de la classe fille. De plus, la classe fille a un accès privilégié aux détails d'implémentation de sa classe mère. Tout fonctionnement interne qui n'est pas soigneusement caché est visible des sous-classes. Si l'objectif est de développer des systèmes faiblement couplés où le moindre changement ne provoque pas une réaction en chaîne qui fait voler en éclats tout le code, il ne faut pas se fier complètement à l'héritage.

Si l'héritage pose autant de soucis, quelle est l'alternative ? Eh bien, nous pouvons construire les comportements souhaités en ayant recours à la composition. Au lieu de créer des classes qui héritent tous leurs talents de la classe mère, nous pouvons les élaborer à partir de composants métier de base. Pour y arriver nous équipons nos objets avec des références vers des objets-fournisseurs de fonctionnalités. Toute classe qui requiert ces fonctionnalités peut les appeler car elles sont encapsulées dans l'objet-fournisseur. En bref, nous préférons définir un objet qui A une caractéristique plutôt qu'un objet qui EST de type donné.

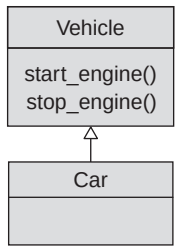
Pour reprendre l'exemple précédent, supposons que nous ayons une méthode qui simule une balade en voiture. La partie clé de cette balade est le démarrage et l'arrêt du moteur :

```
class Vehicle
  # Différentes fonctionnalités des véhicules...
  def start_engine
    # Démarrer le moteur
  end

  def stop_engine
    # Arrêter le moteur
  end
end

class Car < Vehicle
  def sunday_drive
    start_engine
    # Aller se balader et ensuite revenir.
    stop_engine
  end
end
```

La logique de ce code est la suivante : notre voiture a besoin de démarrer et d'arrêter le moteur. Cette fonctionnalité s'appliquera à d'autres véhicules, pourquoi ne pas factoriser le code lié au moteur et le placer dans la classe commune `Vehicle` (voir Figure 1.2).

Figure 1.2

Factoriser le code lié au moteur dans la classe mère



C'est bien, mais tous les véhicules ne sont pas forcément équipés d'un moteur. Une intervention chirurgicale sera nécessaire sur des classes qui représentent des véhicules non motorisés (un vélo ou un bateau à voile). Par ailleurs, à moins de faire attention lors du développement de la classe `Vehicle`, les détails liés au moteur seront disponibles pour la classe `Car`. Après tout, le moteur est géré par la classe `Vehicle`, et l'objet `Car` n'est qu'une sorte de `Vehicle`. Ce n'est pas conforme au principe de séparation des parties variables et statiques.

Toutes ces difficultés peuvent être évitées si le code du moteur est placé dans sa propre classe totalement indépendante et dissociée de la classe parent de `Car`.

```
class Engine
  # Code lié au moteur
  def start
    # Démarrer le moteur
  end

  def stop
    # Arrêter le moteur
  end
end
```

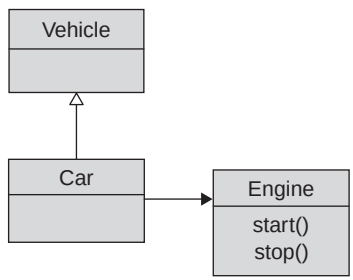
Si chaque objet `Car` contient une référence vers son propre `Engine`, nous pourrions aller nous promener en voiture grâce à la composition.

```
class Car
  def initialize
    @engine = Engine.new
  end

  def sunday_drive
    @engine.start
    # Aller se balader et ensuite revenir.
    @engine.stop
  end
end
```

L'assemblage des fonctionnalités par composition (voir Figure 1.3) donne de nombreux avantages : les fonctions du moteur sont factorisées dans une classe séparée et sont prêtes à être réutilisées (encore une fois par le truchement de la composition !).

Figure 1.3

Assembler une voiture avec la composition



Qui plus est, nous avons également simplifié la classe `Vehicle` en supprimant les fonctions du moteur.

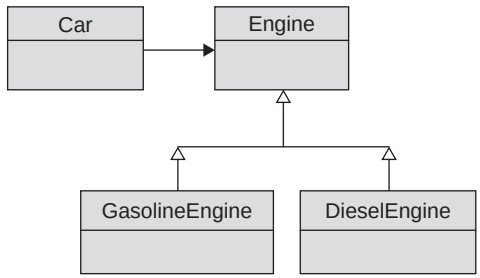
Nous avons aussi amélioré le niveau d'encapsulation : l'extraction des fonctions du moteur offre désormais une séparation nette par interface entre chaque voiture et son moteur. Dans la version initiale, fondée sur l'héritage, tous les détails de l'implémentation du moteur étaient exposés à toutes les méthodes de la classe `Vehicle`. Dans la

nouvelle version, la voiture ne peut accéder à son moteur qu'au travers des fonctions publiques et probablement bien conçues de l'interface de la classe `Engine`.

Nous avons également ouvert une possibilité d'utilisation d'autres types de moteurs. La classe `Engine` pourrait devenir une classe abstraite dont nous pourrions dériver plusieurs implémentations de moteurs, toutes utilisables par notre voiture (voir Figure 1.4).

Figure 1.4

Désormais, la voiture peut avoir des moteurs différents



Cerise sur le gâteau – notre voiture n'est pas condamnée à une seule implémentation de moteur pendant toute sa vie. Les moteurs peuvent être remplacés au moment de l'exécution :

```
class Car
  def initialize
    @engine = GasolineEngine.new
end

  def sunday_drive
    @engine.start
    # Aller se balader et ensuite revenir.
    @engine.stop
  end

  def switch_to_diesel
    @engine = DieselEngine.new
  end
end
```

Déléguer, déléguer, déléguer

Il y a une légère différence fonctionnelle entre notre classe `Car` avec l'objet `Engine` séparé et l'implémentation initiale fondée sur l'héritage. La classe initiale exposait les méthodes publiques `start_engine` et `stop_engine`. Il est évidemment possible de faire de même avec la dernière version de l'implémentation en passant la responsabilité à l'objet `Engine` :

```
class Car
  def initialize
    @engine = GasolineEngine.new
  end

  def sunday_drive
    start_engine
    # Aller se balader et ensuite revenir.
    stop_engine
  end

  def switch_to_diesel
    @engine = DieselEngine.new
  end

  def start_engine
    @engine.start
  end

  def stop_engine
    @engine.stop
  end
end
```

Cette technique simple qui consiste à "refiler le bébé" est connue sous le nom prétentieux de la *délégation*. La méthode `start_engine` est appelée sur l'objet `Car`. L'objet dit "ce n'est pas mon problème" et passe la main à l'objet moteur.

La combinaison de composition et de délégation est une alternative puissante et flexible à l'héritage. On profite de la plupart des avantages de l'héritage tout en gardant beaucoup plus de flexibilité et sans être pénalisé par des effets de bord. Cette facilité n'est pas gratuite. La délégation requiert un appel de méthode supplémentaire lorsqu'un objet passe la main à un autre. Cet appel entraîne un coût sur les performances, mais il reste cependant négligeable dans la plupart des cas.

La délégation a aussi un autre coût : la nécessité d'écrire du code basique – à savoir toutes ces méthodes ennuyeuses comme `start_engine` et `stop_engine` qui ne font que transférer l'appel à l'objet final capable de faire le traitement nécessaire. Heureusement, vous avez entre les mains un livre qui traite des design patterns en Ruby et, comme vous le verrez aux Chapitres 10 et 11, Ruby permet d'éviter d'écrire ces méthodes ennuyeuses.

Vous n'aurez pas besoin de ça

Assez parlé des principes cités par le GoF en 1995. J'aimerais ajouter à cette liste formidable un autre principe que je trouve critique pour développer et livrer de vrais systèmes. Ce principe de conception vient du monde de l'Extreme Programming et est élégamment résumé en *You Ain't Gonna Need It* (YAGNI ou "vous n'en aurez pas

besoin"). Le principe YAGNI stipule que vous ne devez pas implémenter des fonctionnalités ni introduire de la flexibilité si vous n'en avez pas un besoin immédiat. Pourquoi ? Parce qu'il est fort probable que vous n'en ayez pas besoin plus tard non plus.

Un système bien conçu est un système qui s'adapte avec élégance aux corrections de bugs, aux changements des exigences, au progrès continu de la technologie ainsi qu'à des remises à plat inévitables. Selon le principe YAGNI, il faut se concentrer sur des besoins immédiats et développer précisément le niveau de flexibilité dont on est sûr d'avoir besoin. Sans cette certitude, il vaut mieux reporter l'implémentation de la fonctionnalité jusqu'au moment où elle devient nécessaire. Si la fonction n'est pas indispensable, ne l'implémentez pas, consacrez plutôt votre temps et votre énergie à l'écriture des fonctions nécessaires dans l'instant.

À la base du principe YAGNI on trouve un principe souvent vérifié qui dit que nous avons tendance à nous tromper quand nous tentons d'anticiper nos besoins futurs. Lorsqu'on ajoute une nouvelle fonction ou un niveau de flexibilité avant que le besoin ne se manifeste, on fait un double pari.

Premièrement, on parie que cette fonction sera finalement utilisée. Si aujourd'hui vous implémentez votre couche de persistance de manière indépendante de la base de données, vous faites le pari qu'un jour vous aurez à utiliser une autre base de données. Si vous internationalisez l'interface utilisateur, vous pariez qu'un jour vous aurez des utilisateurs à l'étranger. Comme le disait Yogi Berra (selon certaines sources), les prédictions sont difficiles surtout lorsqu'elles concernent le futur. S'il s'avère que vous ne passerez jamais à une autre base de données ou que votre application ne sera pas distribuée ailleurs que dans votre pays d'origine, tout le travail effectué en amont et toute la complexité supplémentaire n'auront servi à rien.

Le second pari que vous faites en développant avant l'heure est encore plus risqué. Lorsque vous implémentez une fonctionnalité ou ajoutez un niveau de flexibilité trop tôt, vous considérez que votre solution est bonne et que vous savez résoudre un problème qui ne s'est pas encore présenté. Vous pariez que votre couche de persistance indépendante de la base de données et installée avec tant d'amour sera adaptée au système effectivement retenu pour remplacer l'ancienne : "Quoi ? Le marketing veut que l'on supporte xyzDB ? Je n'en ai jamais entendu parler !" Vous pariez que votre solution d'internationalisation pourra supporter toute langue que vous serez amené à gérer : "Oh là là ! Je ne me suis pas rendu compte qu'il fallait supporter une langue qui s'écrit de droite à gauche..."

Voici une façon de voir les choses : mis à part la possibilité de recevoir un coup sur la tête, vous ne deviendrez pas plus bête avec le temps. Nous apprenons des choses et gagnons en intelligence chaque jour qui passe, et c'est encore plus vrai dans des projets logiciel. On peut être certain d'avoir une vision plus claire des contraintes, des technologies et de la conception à la fin d'un projet qu'à son commencement. Lorsque vous développez une fonction dont vous n'avez pas encore besoin, vous êtes coupable de programmer bêtement. Attendez le moment où le besoin se manifeste et vous serez probablement en état de mieux comprendre le besoin et la façon d'y répondre.

Le but des design patterns est de rendre vos systèmes plus flexibles et plus adaptables aux changements. Mais, au fil du temps, l'usage des design patterns s'est retrouvé associé à un courant particulièrement virulent de "surengineering" visant à produire du code infiniment flexible au risque de devenir incompréhensible et parfois même défectueux. L'utilisation appropriée des design patterns est l'art de concevoir un système suffisamment flexible pour répondre à vos besoins d'aujourd'hui, pas plus. Un pattern est une technique utile plutôt qu'une fin en soi. Les design patterns peuvent vous aider à développer un système qui fonctionne, mais le système ne fonctionnera pas mieux si vous appliquez toutes les combinaisons imaginables des vingt-trois patterns du GoF. Votre code marchera mieux si vous restez concentré sur les tâches à accomplir aujourd'hui.

Quatorze sur vingt-trois

Les patterns présentés dans l'ouvrage *Design Patterns* sont des outils pour construire des logiciels. Tout comme les outils que l'on achète à la quincaillerie, ils ne sont pas adaptés pour toutes les situations. Certains sont comme votre marteau préféré, indispensables pour toutes vos tâches ; d'autres sont comme le niveau laser que l'on m'a offert pour mon anniversaire, parfaits lorsque vous en avez besoin, ce qui n'arrive que rarement. Dans ce livre, nous allons aborder quatorze des vingt-trois patterns du GoF. En faisant le choix des patterns à couvrir, j'ai essayé de me concentrer sur les pratiques les plus répandues et les plus utiles. Par exemple, je n'imagine pas coder sans utiliser des itérateurs (voir Chapitre 7), j'inclus donc ce pattern sans hésiter. Je me suis également penché vers des patterns qui se transforment lors du passage en Ruby. Une fois de plus, l'itérateur est un bon exemple : il m'est impossible de vivre sans, mais les itérateurs en Ruby diffèrent beaucoup de ceux en Java et C++. Pour vous donner un aperçu de ce qui vous attend, voici une vue d'ensemble des patterns du GoF traités dans ce livre :

- Chacun des patterns essaie de résoudre un problème. Admettons par exemple que votre code fasse toujours la même chose, sauf à l'étape 44. Parfois, l'étape 44 doit

s'exécuter d'une certaine manière et parfois d'une autre. Vous aurez alors probablement besoin du pattern *Template Method.*

- Peut-être est-ce la totalité de l'algorithme qui doit varier et pas seulement l'étape 44. Vous avez une tâche bien définie à effectuer, mais il existe plusieurs façons de le faire. Il conviendrait sans doute d'encapsuler ces techniques – ou algorithmes – dans un objet *Stratégie*.

- Et si vous avez une classe A qui doit rester au courant des événements intervenant dans la classe B ? Mais en évitant le couplage des deux classes car un jour viendra où la classe C (ou même la classe D) verra le jour et exprimera le même besoin. Il faudra alors considérer l'utilisation du pattern *Observateur*.

- Parfois, il faut gérer une collection en tant qu'un seul objet. Il doit être possible de supprimer, déplacer ou copier un fichier isolé ou faire les mêmes opérations sur un répertoire entier. Si vous devez développer une collection qui se comporte comme un objet individuel, vous avez probablement besoin du pattern *Composite*.

- Imaginez que vous écrivez du code pour dissimuler une collection d'objets, mais que vous ne voulez pas la cacher complètement : l'utilisateur doit avoir accès aux objets en séquence sans savoir où ni comment ils sont stockés. Vous avez sans doute besoin du pattern *Itérateur*.

- Parfois, nos instructions doivent être présentées comme une sorte de carte postale : "Chère base de données, quand tu recevras ceci, j'aimerais que tu supprimes la ligne 7843." Les cartes postales sont rares dans le code, mais le pattern *Command* est conçu sur mesure pour ce genre de situation.

- Que faire lorsque vous avez un objet qui fait ce qu'il doit faire mais dont l'interface est totalement inadéquate ? Il peut s'agir ici d'une profonde incohérence ou plus simplement d'un objet qui utilise la méthode `write` sur un objet qui nomme cette méthode `save`. Dans cette situation, le GoF recommande le pattern *Adaptateur*.

- Vous avez peut-être le bon objet sous la main, mais il est là-bas quelque part sur le réseau et vous ne voulez pas que le client ait à s'occuper de son emplacement. Ou peut-être essayerez-vous de retarder le plus possible la création d'un objet ou encore d'implémenter un contrôle d'accès. Dans ces circonstances, optez pour le pattern *Proxy*.

- Il est parfois nécessaire de rajouter certaines responsabilités à un objet à la volée, au moment de l'exécution. Si vous disposez d'un objet avec un certain nombre de

fonctionnalités mais qui de temps en temps doive prendre des responsabilités supplémentaires, il serait judicieux d'utiliser le pattern *Décorateur*.

- Vous pouvez avoir besoin d'un objet unique dont il n'existe qu'une seule instance disponible pour tous les utilisateurs. Cela ressemble fort au pattern *Singleton*.
- Maintenant, imaginez que vous écrivez une classe destinée à être étendue. Pendant que vous êtes en train de coder joyeusement la classe parent, vous vous rendez compte qu'elle doit instancier un nouvel objet. Seulement, la classe fille sera au courant du type d'objet à créer. Vous avez probablement besoin du pattern *Factory Method (Fabrication)*.
- Comment créer des familles d'objets compatibles ? Imaginez que vous ayez un système de conception de voitures. Tous les types de moteurs ne sont pas compatibles avec toutes les sortes de carburant ou les systèmes de refroidissement. Comment s'assurer que l'on ne finisse pas par développer une voiture Frankenstein ? C'est peut-être l'occasion d'utiliser une classe dédiée à la création de ces objets et de l'appeler *Abstract Factory*.
- Supposons que vous instanciez un objet d'une telle complexité qu'un volume important de code soit nécessaire pour gérer sa construction. Ou, pire encore, le processus de la construction est variable selon les circonstances. Vous avez probablement besoin du pattern *Builder*.
- Avez-vous déjà eu le sentiment de ne pas utiliser le bon langage de programmation pour résoudre votre problème ? Cela pourrait paraître fou, mais peut-être que vous devriez vous arrêter et développer un *Interpréteur* pour le langage spécifique capable de fournir une solution simple.

Patterns en Ruby ?

Voilà en quelques lignes la théorie des patterns[1]. Mais pourquoi Ruby ? Ne s'agit-il pas d'un quelconque langage de script seulement approprié pour des tâches d'administration système et des interfaces graphiques Web ?

En un mot : non.

Ruby est un langage orienté objet, élégant et universel. Il n'est pas parfait pour toutes les situations – par exemple, si vous avez besoin de très haute performance, il faut, au

1. Évidemment, je ne fais que gratter la surface d'un sujet vaste et passionnant. Jetez un œil sur l'annexe B, Aller plus loin, pour plus d'information.

moins pour l'instant, choisir un autre langage. Toutefois, Ruby est plus que convenable pour un grand nombre de tâches. Ce langage a une syntaxe concise mais très expressive et incorpore un modèle de programmation riche et sophistiqué.

Vous verrez dans les prochains chapitres que Ruby a sa propre façon de faire les choses, ce qui change notre approche des différents problèmes de programmation, y compris ceux abordés par des patterns classiques du GoF. Par conséquent, il n'est pas étonnant que la combinaison de Ruby et des design patterns classiques mène vers des développements nouveaux et non traditionnels. Parfois, Ruby est suffisamment différent pour offrir des solutions totalement innovantes. À ce propos, trois nouveaux patterns attirent l'attention avec la popularité récente de Ruby. C'est la raison pour laquelle je conclus le catalogue des patterns par les modèles suivants :

- *Internal Domain–Specific Language (DSL)*, une technique très dynamique pour construire des petits langages spécialisés ;
- *Méta-programmation*, une technique de création dynamique des classes et des objets au moment de l'exécution ;
- *Convention plutôt que configuration*, un remède contre les maux de configuration (principalement XML).

Commençons...

2

Démarrer avec Ruby

J'ai découvert Ruby grâce à mon fils de 8 ans et à son amour pour une gentille souris jaune électriquement chargée[1]. À l'époque, en 2002, mon fils passait son temps libre à jouer à un jeu vidéo dont le but était de trouver et d'apprivoiser différentes créatures magiques, y compris le rongeur énergétique. Un jour, j'ai eu l'impression de voir une ampoule s'allumer au-dessus de sa tête et j'imaginais ses pensées : "Mon papa est programmeur. Le jeu auquel je joue, le truc qui parle des îles magiques et des êtres merveilleux, est un programme. Mon papa fait des programmes. Mon papa peut m'apprendre à faire un jeu !"

Eh bien, peut-être ! Après une dose de harcèlement et de pleurnicheries que seuls les parents de jeunes enfants peuvent réellement comprendre, j'ai entrepris d'apprendre la programmation à mon fils. Tout d'abord, nous avions besoin d'un langage de programmation simple, clair et facilement compréhensible. Après une courte recherche j'ai trouvé Ruby. Mon fils, comme le font souvent les enfants, est rapidement passé à autre chose, mais Ruby avait trouvé un nouvel adepte. De fait, c'était un langage propre, clair et simple – parfait pour apprendre. Mais, en tant que développeur professionnel qui a conçu des systèmes dans toutes sortes de langages allant de l'assembleur à Java, j'ai vu davantage : Ruby est concis, sophistiqué et drôlement puissant.

Ruby est également très classique. Les composants de base du langage sont des vieux engrenages connus de tous les programmeurs. Ruby possède tous les types de données courants : des chaînes de caractères, des entiers, des nombres à virgule flottante ainsi que des tableaux et nos vieux amis vrai et faux. Les bases de Ruby sont familières et ordinaires, mais la façon dont le langage est structuré au plus haut niveau nous apporte une joie inattendue.

1. Heureusement que la folie des Pokémon s'est calmée depuis.

Si vous maîtrisez déjà les bases de Ruby, si vous avez déjà écrit quelques classes et savez comment extraire le troisième caractère d'une chaîne ou comment calculer 2 puissance 437, vous pouvez passer directement au Chapitre 3. Cette section sera toujours là pour vous secourir si le besoin s'en fait sentir.

Ce chapitre vous est destiné si vous débutez en programmation Ruby. Le but de cette section est de vous donner le plus brièvement possible un aperçu des notions de base du langage. Au fond, Alan Turing avait raison lorsqu'il disait : "Une fois acquis un certain niveau de complexité, tous les langages de programmation deviennent équivalents." Si vous connaissez un autre langage répandu, les bases de Ruby ne vous poseront aucun problème, vous allez "réapprendre" les choses que vous connaissez déjà. J'espère qu'à la fin de ce chapitre vous connaîtrez du langage juste ce qu'il faut pour devenir dangereux.

Ruby interactif

La façon la plus simple d'exécuter du code Ruby[1] consiste à utiliser la console interactive irb. Après avoir démarré irb, vous pouvez saisir du code Ruby et voir le résultat instantanément. Pour lancer irb, il suffit de taper `irb` dans le terminal. L'exemple ci-après démarre irb et additionne 2 et 2 à l'aide de Ruby :

```
$ irb
irb(main):001:0> 2+2
=> 4
irb(main):002:0>
```

La console interactive Ruby est un moyen formidable pour faire des essais à petite échelle. Rien n'aide plus à explorer un nouveau langage que la possibilité de voir le résultat immédiatement.

Afficher Hello World

Maintenant que Ruby est opérationnel, l'étape suivante tombe sous le sens : écrire votre premier programme. Voici l'incontournable "Hello World" en Ruby :

```
#
# Premier programme traditionnel pour tous les langages
#
puts('hello world')
```

1. Voyez l'annexe A si vous avez besoin d'instructions pour installer Ruby sur votre système.

Vous pouvez simplement démarrer irb et entrer ce code de manière interactive. Une autre alternative est d'écrire le programme avec un éditeur de texte et de l'enregistrer dans un fichier, par exemple sous le nom `hello.rb`. Ensuite, vous pouvez exécuter votre programme à l'aide de l'interpréteur Ruby, commodément appelé `ruby` :

```
$ ruby hello.rb
```

Les deux techniques donnent un résultat identique :

```
hello world
```

Rien qu'en lisant le programme hello world, vous pouvez apprendre beaucoup sur un langage de programmation. Par exemple, on découvre que la méthode `puts` affiche des choses. On voit également que les commentaires commencent par le caractère `#` et continuent jusqu'à la fin de la ligne. Les commentaires peuvent occuper toute la ligne, comme dans l'exemple ci-dessus, ou on peut également les placer après le code :

```
puts('hello world') # Afficher bonjour
```

L'absence de point-virgule à la fin de chaque instruction est un autre point remarquable. En règle générale, une instruction Ruby se termine par un saut de ligne. Les programmeurs Ruby ont tendance à utiliser des points-virgules uniquement pour séparer des instructions multiples sur la même ligne, et ce dans les rares cas où ils décident d'entasser plusieurs instructions sur une ligne :

```
#
# Usage valide, mais atypique d'un point-virgule en Ruby
#
puts('hello world');
#
# Un peu plus de rigueur. Utilisation toujours rare d'un point-virgule
#
puts('hello '); puts('world')
```

L'analyseur syntaxique de Ruby est suffisamment intelligent pour poursuivre son analyse sur la ligne suivante lorsqu'une instruction est clairement inachevée. Par exemple, le code ci-après fonctionne bien, car l'analyseur syntaxique déduit de l'opérateur `+` à la fin de la ligne que l'instruction continue sur une deuxième ligne :

```
x = 10 +
  20 + 30
```

Une instruction peut indiquer explicitement avec une barre oblique inverse qu'elle s'étend sur la ligne suivante :

```
x = 10 \
  + 10
```

On voit ici émerger un principe récurrent de la philosophie Ruby qui consiste à aider l'utilisateur lorsqu'il a besoin d'assistance et à s'effacer en toute autre circonstance. Selon cette philosophie, Ruby permet d'omettre les parenthèses si elles n'aident pas à clarifier une liste d'arguments :

```
puts 'hello world'
```

Pour des raisons de clarté, la plupart des exemples de ce livre incluent les parenthèses dans les appels de méthodes sauf si aucun argument n'est fourni, et dans ce cas les parenthèses vides sont omises.

Dans le programme "hello world" nous avons entouré la chaîne de caractères de guillemets simples, mais les guillemets doubles peuvent tout aussi bien être utilisés :

```
puts("hello world")
```

Le résultat obtenu avec des guillemets simples ou doubles est le même, mais avec une petite subtilité. Les guillemets simples ont pour effet d'afficher littéralement ce que vous voyez car Ruby ne fait que très peu d'interprétations sur de telles chaînes de caractères. Ce n'est pas le cas des guillemets doubles, qui provoquent un traitement classique en amont : `\n` est converti en un caractère de saut de ligne, `\t` devient une tabulation. Si la chaîne `'abc\n'` compte cinq caractères (les deux derniers sont la barre oblique inverse et la lettre "n"), la longueur de la chaîne `"abc\n"` n'est que de quatre caractères (le dernier caractère étant le saut de ligne).

Finalement, vous avez probablement remarqué que la chaîne `'hello world'` que nous avons passée à l'opérateur `puts` n'inclut pas de `\n`. Pourtant, cet opérateur a ajouté un saut de ligne à la fin du message. En réalité, l'opérateur `puts` est assez intelligent. Il rajoute un saut de ligne au texte de sortie s'il en manque un. Ce comportement n'est pas forcément souhaité pour le formatage de précision, mais il est parfaitement adapté pour les exemples que vous trouverez dans ce livre.

Variables

Les noms des variables ordinaires en Ruby commencent par une lettre minuscule ou un tiret bas (nous verrons quelques variables inhabituelles plus tard)[1]. Le premier caractère peut être suivi par des lettres minuscules ou majuscules, des tirets bas ainsi que des chiffres. Les noms des variables sont sensibles à la casse et la seule limite à la longueur

1. Dans la plupart des cas, Ruby considère le tiret bas comme un caractère minuscule.

des variables Ruby est votre imagination. Tous les noms ci-après représentent des noms de variables valides :

- `max_length`
- `maxLength`
- `numberPages`
- `numberpages`
- `a_very_long_variable_name`
- `_flag`
- `column77Row88`
- `___`

Les deux noms `max_length` et `maxLength` évoquent un point important : les noms en casse mixte sont parfaitement acceptés en Ruby et, pourtant, les développeurs Ruby ont tendance à ne pas les utiliser. La pratique la plus répandue parmi les programmeurs Ruby bien élevés est de séparer_des_mots_par_des_tirets_bas. Puisque les noms des variables sont sensibles à la casse, `numberPages` et `numberpages` sont deux variables différentes. Enfin, le dernier nom dans la liste comprend seulement trois tirets bas. C'est certes une pratique valide mais à éviter[1].

Assemblons maintenant nos chaînes de caractères et nos variables :

```
first_name = 'russ'
last_name = 'olsen'
full_name = first_name + ' ' + last_name
```

Cet exemple illustre trois affectations basiques : la chaîne de caractères `'russ'` est attribuée à la variable `first_name`, la valeur `'olsen'` est attribuée à `last_name` et `full_name` reçoit la concaténation de mon prénom et de mon nom séparés par une espace.

Vous avez peut-être remarqué qu'aucune des variables n'est déclarée dans cet exemple. Rien ne déclare que la variable `first_name` est et restera toujours une chaîne de caractères. Ruby est un langage dynamiquement typé, ce qui signifie que les variables ne possèdent pas de type fixe. En Ruby vous pouvez faire apparaître un nom de variable comme par magie et lui affecter une valeur. La variable adoptera le type de la valeur

1. Une autre bonne raison de ne pas nommer des variables uniquement avec des tirets bas est le fait qu'irb a dégainé plus vite que vous. irb affecte à la variable _ (un tiret bas) la valeur de la dernière expression évaluée.

qu'elle reçoit. Non seulement cela, mais la variable peut contenir des valeurs radicalement différentes aux différents moments de l'exécution du programme. Au début du programme, la valeur de `pi` peut être le nombre `3,14159` ; ensuite, au sein du même programme la valeur peut prendre une référence vers un algorithme mathématique complexe et, encore plus tard, elle peut devenir une chaîne de caractères `"apple"`. Dans ce livre nous allons revisiter le typage dynamique à plusieurs reprises (et nous commencerons au chapitre suivant, si vous êtes impatient). Pour l'instant, il faut retenir que les variables adoptent les types de leurs valeurs.

En plus des variables habituelles que nous venons de voir, Ruby supporte les constantes. Une constante ressemble à une variable sauf que son nom commence par une lettre majuscule :

```
POUNDS_PER_KILOGRAM = 2.2
StopToken = 'end'
FACTS = 'Death and taxes'
```

Le principe d'une constante est de recevoir une valeur qui ne change pas. Ruby n'est pas particulièrement rigoureux en ce qui concerne ce comportement. On peut modifier la valeur d'une constante mais au prix d'un avertissement :

```
StopToken = 'finish'
(irb):2: warning: already initialized constant StopToken
```

Par précaution, vous devez éviter de changer les valeurs des constantes.

Fixnums et Bignums

Vous ne serez pas étonné d'apprendre que Ruby supporte les opérations arithmétiques. En Ruby on peut additionner, soustraire, multiplier et diviser comme d'habitude :

```
x = 3
y = 4
sum = x+y
product = x*y
```

En Ruby, un certain nombre de règles s'appliquent aux nombres. Il y a deux types de base : des entiers et des nombres à virgule flottante. Évidemment, les entiers ne contiennent pas de partie décimale : 1 ; 3 ; 6 ; 23 ; –77 et 42 sont tous des entiers, alors que 7.5 ; 3.14159 et –60000.0 sont des nombres à virgule flottante.

La division des entiers en Ruby ne réserve aucune surprise : divisez deux entiers et vous obtiendrez un entier, la partie décimale sera tronquée (et non pas arrondie !) :

```
6/3   # donne 2
7/3   # fait toujours 2
8/3   # 2 une fois de plus
9/3   # et finalement 3 !
```

Les entiers de taille raisonnable – tous ceux qui peuvent être représentés sur 31 octets – sont de type `Fixnum`. Les entiers plus grands sont de type `Bignum`. Un `Bignum` peut contenir tout nombre arbitrairement gigantesque. Étant donné qu'on passe d'un type à l'autre de façon quasiment transparente, vous pouvez considérer qu'on ne fait aucune distinction entre les deux :

```
2                     # Un Fixnum
437                   # Un Fixnum
2**437                # Très certainement un grand Bignum
1234567890            # Encore un Bignum
1234567890/1234567890 # Divisez deux Bignums, et vous obtenez 1
                      # un Fixnum
```

Ruby fournit les astuces d'affectations classiques permettant de raccourcir des expressions. Par exemple, `a = a+1` devient `a += 1` :

```
a = 4
a+ = 1  # est devenu 5
a- = 2  # est devenu 3
a* = 4  # est devenu 12
a/ = 2  # est devenu 6
```

Malheureusement (*NdT* : ou heureusement !), en Ruby il n'y a pas d'opérateurs d'incrémentation (++) et de décrémentation (--).

Nombres à virgule flottante

Si seulement le monde était aussi précis que les entiers ! Mais pour faire face à la complexité du monde réel Ruby fournit des nombres à virgule flottante ou, en terminologie Ruby, des `floats`. Un `float` se distingue facilement, c'est un nombre avec une partie décimale :

```
3.14159
-2.5
6.0
0.0000000111
```

On peut additionner, soustraire et multiplier des `floats` pour obtenir les résultats attendus. Les `floats` obéissent aux règles traditionnelles de la division :

```
2.5 + 3.5 # egale 6.0
0.5*10 # egale 5.0
8.0/3.0 # egale 2.66666666
```

Il n'y a pas de types primitifs ici

Vous n'êtes pas obligé de me croire sur parole en ce qui concerne les types de toutes ces espèces de nombres. Demandez plutôt à Ruby en utilisant la méthode `class` :

```
7.class             # Renvoie class Fixnum
888888888888.class  # Renvoie class Bignum
3.14159.class       # Renvoie class Float
```

Cette syntaxe peut paraître quelque peu étrange, mais c'est un aperçu d'un aspect profond et important : en Ruby, tout – absolument tout – est objet. Lorsque nous écrivons `7.class`, nous utilisons la syntaxe bien connue orientée objet pour appeler la méthode `class` sur un objet. Dans ce cas précis l'objet représente le nombre sept. Les nombres en Ruby disposent d'un large éventail de méthodes :

```
3.7.round     # renvoie 4.0
3.7.truncate  # renvoie 3
-123.abs      # renvoie 123
1.succ        # Successeur, ou le nombre suivant, 2
```

Contrairement à Java, à C# et à de nombreux langages répandus, Ruby n'a pas de types primitifs. Ce sont des objets purs et durs. Le fait que tout en Ruby est objet conditionne en grande partie l'élégance du langage. Par exemple, l'orientation objet universelle de Ruby est le secret derrière la conversion facile entre `Fixnum` et `Bignum`.

Si on trace la hiérarchie de classes d'un objet Ruby quelconque vers sa classe parent, puis vers la classe parent du parent et tous les autres ancêtres, on finira par atteindre la classe `Object`. Grâce à cette ascendance commune, tout objet Ruby hérite d'un minimum de méthodes, une sorte de kit de survie. La méthode `class` que nous avons appelée ci-dessus provient de cette source. On peut également découvrir si l'objet est une instance d'une classe donnée :

```
'hello'.instance_of?(String)  # vrai
```

Ou s'il est `nil` :

```
·hello1.nil?                  # faux
```

L'une des méthodes de la classe `Object` les plus utilisées est probablement `to_s`, qui retourne une représentation de l'objet sous forme d'une chaîne de caractères. C'est une méthode équivalente de la méthode Java `toString` avec le nom commodément raccourci :

```
44.to_s       # retourne une chaîne de deux caractères '44'
'hello'.to_s  # une conversion pas très impressionnante
              # qui retourne 'hello'
```

L'orientation objet totale de Ruby a aussi certaines implications sur les variables. Puisque tout en Ruby est objet, il n'est pas tout à fait correct d'affirmer que l'instruction `x = 44` affecte la valeur `44` à la variable `x`. En réalité, la variable `x` reçoit une référence vers l'objet qui représente le nombre qui suit `43`.

Mais, parfois, il n'y a pas d'objet

Si tout est objet, qu'arrive-t-il si l'on n'a pas vraiment d'objet ? Dans ce cas Ruby fournit un objet spécial qui représente l'idée de ne pas avoir d'objet, d'être complètement dépourvu d'objet. Cette valeur spéciale est `nil`.

Dans la section précédente, nous avons vu que tout en Ruby est objet, ce qui est exact : `nil` est un vrai objet Ruby tout comme `"hello world"` ou `43`. On peut par exemple obtenir la classe de `nil` :

```
puts(nil.class)
```

Le résultat est prévisible :

```
NilClass
```

Malheureusement, `nil` est prédestiné à vivre sa vie tout seul : il n'y a qu'une unique instance de `NilClass` (appelée `nil`), et aucune autre instance de `NilClass` ne peut être créée.

Vérité, mensonges et nil

Ruby supporte la panoplie classique des opérateurs booléens. Nous pouvons par exemple déterminer si deux expressions sont égales, si l'une est inférieure ou supérieure à l'autre.

```
1 == 1            # vrai
1 == 2            # faux
'russ' == 'smart' # malheureusement, faux
(1 < 2)           # vrai
(4 > 6)           # et non
a = 1
b = 10000
(a > b)           # pas question
```

Nous avons aussi inférieur ou égal et son cousin supérieur ou égal :

```
(4 >= 4) # oui!
(1 <= 2) # vrai aussi
```

Tous les opérateurs de comparaison sont évalués comme l'un de ces deux objets – `true` ou `false`. Tout comme `nil`, `true` et `false` sont des instances uniques de leurs classes respectives : `true` est la seule instance de `TrueClass` et `false` est la seule instance de (vous l'avez deviné) `FalseClass`. Étrangement, `TrueClass` et `FalseClass` sont des sous-classes directes d'`Object`. On aurait pu s'attendre à trouver une classe `BooleanClass` quelque part mais, hélas, elle n'existe pas !

Ruby offre aussi un opérateur `and`. Il en a même plusieurs :

```
(1 == 1) and (2 == 2) # vrai
(1 == 1) and (2 == 3) # faux
```

On pourrait également écrire :

```
(1 == 1) && (2 == 2) # vrai
(1 == 1) && (2 == 3) # faux
```

Les résultats des deux sont équivalents. Généralement, les opérateurs `and` et `&&` sont des synonymes[1]. Les opérateurs `or` et `||` sont assortis avec `and` et `&&` et provoquent un résultat attendu[2] :

```
(1 == 1) or (2 == 2) # oui
(2 == 1) || (7 > 10) # non
(1 == 1) or (3 == 2) # oui
(2 == 1) || (3 == 2) # non
```

Enfin, Ruby fournit un opérateur classique `not` et son jumeau `!` :

```
not (1 == 2)  # vrai
! (1 == 1)    # faux
not false     # vrai
```

Il faut bien tenir compte du fait qu'en Ruby toute expression peut être évaluée de manière booléenne. Nous pouvons mélanger des chaînes de caractères avec des entiers ou encore des dates pour obtenir un booléen. Les règles d'évaluation sont très simples : `false` et `nil` sont évalués à `false`. Toute autre expression provoque le résultat `true`. Par conséquent, les expressions suivantes sont parfaitement valides :

```
true and 'fred' # vrai, puisque 'fred' n'est pas nil ni false
'fred' && 44    # vrai, puisque 'fred' et 44 sont tous les deux à true
nil || false    # faux, puisque nil et false s'évaluent à false
```

1. Pas complètement. L'opérateur `&&` a une priorité plus forte par rapport à `and`. La même règle s'applique à `||` et `or`.
2. Il existe en Ruby des opérateurs `&` et `|` – remarquez qu'ils ne consistent qu'en un seul caractère. Ce sont des opérateurs logiques bit à bit, qui sont très utiles dans certaines situations, mais probablement pas dans vos instructions quotidiennes de comparaison.

Si vous venez du monde de C ou C++, vous trouverez choquant le fait que zéro en Ruby s'évalue à `true` dans les expressions booléennes (car zéro n'est pas `nil` ni `false`). Aussi surprenant que cela puisse paraître, l'expression

```
if 0
  puts('Zero is true!')
end
```

affiche

```
Zero is true!
```

Décisions, décisions

L'exemple précédent était un aperçu de l'instruction `if` qui vient avec son `else` facultatif :

```
age = 19
if (age >= 18)
  puts('You can vote!')
else
  puts('You are too young to vote.')
end
```

Comme vous pouvez le voir, chaque instruction `if` doit être obligatoirement fermée par `end`. Dans le cas où il y a plus d'une condition, on peut utiliser `elsif` :

```
if(weight < 1)
  puts('very light')
elsif(weight < 10)
  puts('a bit of a load')
elsif(weight < 100)
  puts('heavy')
else
  puts('way too heavy')
end
```

Il faut noter que le mot clé `elsif` est un seul mot de cinq lettres. Ce n'est pas `else if`, ni `elseif` et encore moins `elif`.

Ruby essaie toujours de rendre le code le plus concis possible. Vu que les parenthèses autour des instructions `if` et `elsif` n'ajoutent pas de valeur au code, elles sont facultatives :

```
if weight < 1
  puts('very light')
elsif weight < 10
  puts('a bit of a load')
```

```
elsif weight < 100
  puts('heavy')
else
  puts('way too heavy')
end
```

Il existe un idiome particulier si vous avez à prendre une décision quant à l'exécution d'une seule instruction Ruby. Dans ce cas vous pouvez tout simplement appliquer l'opérateur `if` à la fin de la ligne :

```
puts('way too heavy') if weight >= 100
```

L'opérateur `unless` est l'inverse de l'opérateur `if` : le corps de l'instruction ne s'exécute que si la condition est évaluée à `false`. Tout comme avec l'opérateur `if`, `unless` peut avoir une forme longue :

```
unless weight < 100
  puts('way too heavy')
end
```

ou une forme courte :

```
puts('trop lourd') unless weight < 100
```

Boucles

Ruby possède deux sortes de boucles. Tout d'abord, il y a la boucle `while` classique, qui doit toujours se terminer par `end` comme une expression `if`. La boucle suivante

```
i = 0 while i < 4
  puts("i = #{i}")
  i = i + 1
end
```

affiche ceci :

```
i = 0
i = 1
i = 2
i = 3
```

Le frère ennemi de `while` est `until`, qui est plus ou moins identique à `while` avec la seule différence que la boucle continue à être exécutée jusqu'au moment où la condition devient `true`. L'exemple précédent peut être écrit de façon suivante :

```
i = 0
until i >= 4
  puts("i = #{i}")
  i = i + 1
end
```

Une boucle `for` en Ruby peut être utilisée par exemple pour accéder aux éléments d'un tableau en séquence :

```
array = ['first', 'second', 'third']
for element in array
  puts(element)
end
```

Étonnamment, les boucles `for` sont rares dans les vrais programmes Ruby. Un programmeur Ruby est plus susceptible d'écrire ce code équivalent :

```
array.each do |x|
  puts(x)
end
```

Cette boucle à l'allure bizarre est décrite de manière détaillée au Chapitre 7. Pour l'instant, voyez la syntaxe `each` comme une manière alternative d'écrire des boucles `for`. Si la boucle doit être interrompue, on peut utiliser l'instruction `break` :

```
names = ['george', 'mike', 'gary', 'diana']
names.each do |name|
  if name == 'gary'
    puts('Break!')
    break
  end
  puts(name)
end
```

Si vous exécutez ce code, il n'affichera jamais `gary` :

```
george
mike
Break!
```

Enfin, on peut omettre l'exécution d'une itération à l'aide de l'instruction `next` :

```
names.each do |name|
  if name == 'gary'
    puts('Next!')
    next
  end
  puts(name)
end
```

Ce code n'affichera jamais `gary` mais continuera l'exécution de la boucle :

```
george
mike
Next!
diana
```

Plus de détails sur les chaînes de caractères

Essayons de nous familiariser davantage avec les chaînes de caractères puisque nous les utilisons déjà. Comme nous l'avons vu précédemment, les chaînes de caractères peuvent être construites à la fois avec des guillemets simples et des guillemets doubles :

```
first = 'Mary had'
second = " a little lamb"
```

Nous avons également appris que le signe plus est un opérateur de concaténation, donc

```
poem = first + second
```

s'évalue à :

```
Mary had a little lamb
```

Les chaînes disposent de tout un éventail de méthodes. On peut par exemple obtenir la longueur d'une chaîne :

```
puts(first.length) # Affiche 8
```

On peut aussi convertir une chaîne en majuscules ou en minuscules :

```
puts(poem.upcase)
puts(poem.downcase)
```

Ce code affiche

```
MARY HAD A LITTLE LAMB
mary had a little lamb
```

Le comportement des chaînes de caractères en Ruby ressemble à celui des tableaux : on peut affecter un caractère précis à une chaîne par son index, comme un élément d'un tableau. Si l'on exécute

```
poem[0] = 'G'
puts(poem)
```

le résultat sera un poème très différent :

```
Gary had a little lamb
```

De même, on peut accéder à des caractères particuliers dans une chaîne, mais avec un petit souci : Ruby ne possède pas de type de caractère spécial. Par conséquent, si l'on extrait des caractères d'une chaîne en Ruby, on obtient un nombre entier, le code du caractère. Voyez l'exemple suivant :

```
second_char = poem[1] # le caractère second_char est égal à 97,
                      # le code ASCII de la lettre 'a'
```

Heureusement, on peut aussi réinsérer des caractères et, finalement, nous n'avons peut-être pas causé de dommages :

```
poem[0] = 67 # 67 est le code ASCII de la lettre 'C'
```

Maintenant, le propriétaire de l'agneau est Cary.

Les chaînes de caractères entourées de guillemets doubles ont en Ruby une caractéristique spéciale que nous allons souvent rencontrer dans les exemples de ce livre. En plus de remplacer les `\n` par des sauts de ligne et les `\t` par des tabulations, lorsque l'interpréteur Ruby trouve `#{expression}` à l'intérieur d'une chaîne entre doubles guillemets, il remplace l'expression par sa valeur. Par exemple, si l'on affecte une valeur à la variable `n`

```
n = 42
```

on peut simplement l'insérer dans une chaîne de caractères

```
puts("The value of n is #{n}.")
```

pour obtenir

```
The value of n is 42.
```

Cette fonctionnalité (qui s'appelle l'interpolation de chaînes de caractères) ne se limite pas à une seule expression par chaîne, et les expressions ne se limitent pas simplement à des noms de variables. Nous pouvons très bien écrire

```
city = 'Washington'
temp_f = 84
puts("The city is #{city} and the temp is #{5.0/9.0 * (temp_f-32)} C°")
```

le résultat affiché sera

```
The city is Washington and the temp is 28.8888888888889 C°
```

Les guillemets simples sont parfaits pour des chaînes de caractères relativement courtes, qui tiennent sur une ligne, mais ils sont peu commodes pour des expressions à plusieurs lignes. Pour remédier à cela, Ruby fournit un autre moyen d'exprimer des chaînes de caractères littérales :

```
a_multiline_string = %Q{
The city is #{city}.
The temp is #{5.0/9.0 * (temp_f-32)} C°
```

Dans cet exemple, tout ce qui se trouve entre `%Q{` et `}` est une chaîne de caractères. Si votre chaîne commence par `%Q {` comme ci-dessus, Ruby la considère comme une chaîne entourée des doubles guillemets et fait toute l'interprétation en conséquence. Si

l'on utilise `%q{` (remarquez que "q" est minuscule), le texte n'est traité que de manière minimaliste, correspondant à une chaîne entre des guillemets simples[1].

Enfin, si vous venez du monde de Java ou C#, un sérieux piège conceptuel vous attend en Ruby. Les chaînes en C# et Java sont *immuables* : une fois créée, la chaîne de caractères ne peut jamais être modifiée. Ce n'est pas le cas de Ruby. En Ruby, une chaîne de caractères est modifiable à tout instant. Créons deux références à la même chaîne de caractères pour illustrer notre propos :

```
name = 'russ'
first_name = name
```

Si l'on écrivait du code Java ou C#, on pourrait manipuler `first_name` à l'infini, en étant certain que sa valeur ne peut jamais être modifiée. Contrairement à ces langages, si l'on changeait la valeur de `name` :

```
name[0] = 'R'
```

on modifierait également `first_name`, qui n'est qu'une référence au même objet de type chaîne de caractères. En affichant les deux variables

```
puts(name)
puts(first_name)
```

on obtient la valeur modifiée :

```
Russ
Russ
```

Symboles

Une polémique autour des avantages des chaînes immuables existe depuis bien longtemps. Les chaînes de caractères étaient modifiables en C et C++, puis immuables en Java et C#, et sont ensuite redevenues modifiables en Ruby. Les chaînes transformables ont sans aucun doute des avantages, mais le fait de les rendre modifiables laisse une lacune évidente : que faire s'il faut représenter un identificateur interne plutôt que des données ?

---

1. En vérité, nous disposons de beaucoup plus d'options. Nous pouvons par exemple choisir des parenthèses "`()`" ou des chevrons "`<>`" au lieu des accolades que j'utilise pour délimiter des chaînes. Ainsi, `%q<une chaîne>` est une chaîne de caractères parfaitement valide. On peut utiliser un caractère spécial de son choix pour commencer et terminer un chaîne, par exemple `%Q-une chaîne-`.

Ruby fournit dans ce cas une classe spéciale, le symbole. Les symboles Ruby sont des identificateurs immuables. Ils commencent toujours par un deux-points :

- `:a_symbol`
- `:an_other_symbol`
- `:first_name`

Si vous n'êtes pas habitué aux symboles, ils peuvent paraître étranges au début. Il suffit de retenir que les symboles sont plus ou moins des chaînes de caractères immuables que les programmeurs Ruby utilisent en tant qu'identificateurs.

Tableaux

Créer des tableaux Ruby se fait simplement en tapant au clavier une paire de crochets ou `Array.new` :

```
x = []                          # Un tableau vide
y = Array.new                   # Encore un
a = ['neo', 'trinity', 'tank']  # Un tableau de trois éléments
```

Les éléments d'un tableau Ruby sont énumérés à partir de zéro :

```
a[0] # neo
a[2] # tank
```

On peut se renseigner sur le nombre des éléments d'un tableau à l'aide des méthodes `length` ou `size`. Les deux sont équivalentes :

```
puts(a.length)  # est 3
puts(a.size)    # est 3 aussi
```

N'oubliez pas que le nombre d'éléments d'un tableau Ruby n'est pas fixe. Un tableau croît dynamiquement si on lui affecte un nouvel élément à la fin :

```
a[3] = 'morpheus'
```

Maintenant, le tableau compte quatre éléments.

Si l'on ajoute un élément au tableau au-delà de sa longueur courante, Ruby crée automatiquement les éléments intermédiaires et leur attribue la valeur `nil`. Donc, le résultat du code suivant

```
a[6] = 'keymaker'
puts(a[4])
puts(a[5])
puts(a[6])
```

est

```
nil
nil
keymaker
```

L'opérateur << fournit un moyen simple d'ajouter un élément à la fin d'un tableau :

```
a << 'mouse'
```

Le principe du typage dynamique de Ruby s'applique également aux tableaux. En Ruby, les tableaux ne sont pas limités à un seul type d'élément. Dans un seul tableau nous pouvons mélanger à volonté différents types d'objets :

```
mixed = ['alice', 44, 62.1234, nil, true, false]
```

Enfin, vu que les tableaux sont des objets normaux[1], ils bénéficient d'une riche palette de méthodes. Par exemple, on peut les trier par ordre ascendant :

```
a = [77, 10, 120, 3]
a.sort # renvoie [3, 10, 77, 120]
```

ou les inverser :

```
a = [1, 2, 3]
a.reverse # renvoie [3, 2, 1]
```

Il est important de savoir que les méthodes `sort` et `reverse` ne modifient pas le tableau original, elles retournent une nouvelle instance triée. Si vous avez besoin de trier le tableau original, optez pour les méthodes `sort!` et `reverse!` :

```
a = [77, 10, 120, 3]
a .sort!    # a est maintenant [3, 10, 77, 120]
a .reverse! # a est maintenant [120, 77, 10, 3]
```

La convention de nommage spécifiant qu'une `méthode` ne modifie pas l'objet original tandis qu'une `méthode!` opère sur l'objet original n'est pas limitée aux tableaux. Elle est fréquemment (mais malheureusement pas encore universellement) employée dans le langage Ruby.

Tableaux associatifs

Un tableau associatif Ruby est un ami intime d'un tableau. On peut considérer les tableaux associatifs comme des tableaux qui acceptent tout objet en tant que clé de

1. Nous savons que les tableaux sont des objets Ruby puisque (tous ensemble !) en Ruby tout est objet.

hachage. Mais, contrairement aux tableaux, les tableaux associatifs ne sont pas ordonnés. On crée un tableau associatif avec une paire d'accolades :

```
h = {}
h['first_name'] = 'Albert'
h['last_name'] = 'Einstein'
h['first_name']   # est 'Albert'
h['last_name']    # est Einstein
```

Il existe une syntaxe raccourcie pour initialiser un tableau associatif. Le même tableau associatif peut être défini de manière suivante :

```
h = {'first_name' => 'Albert', 'last_name' => 'Einstein'}
```

Les symboles font d'excellentes clés de hachage. Ainsi, l'exemple précédent peut être amélioré ainsi :

```
h = {:first_name => 'Albert', :last_name => 'Einstein'}
```

Expressions régulières

Le dernier type Ruby que nous allons étudier est l'expression régulière. Une expression régulière en Ruby est placée entre deux barres obliques :

```
/old/
/Russ|Russell/
/.*/
```

Les expressions régulières permettent de faire des choses extrêmement complexes, mais les principes de base de leur fonctionnement sont vraiment très simples. Même des connaissances superficielles en la matière vous aideront énormément[1]. En deux lignes, une expression régulière est un modèle qui correspond ou pas à une chaîne de caractères donnée. Par exemple, la première des trois expressions régulières ci-dessus correspondra uniquement à la chaîne `'old'`, tandis que la deuxième correspondra aux deux variantes de mon prénom. La troisième expression correspondra à n'importe quelle chaîne de caractères.

L'opérateur `=~` de Ruby permet de vérifier qu'une expression donnée correspond à une chaîne de caractères particulière. L'opérateur `=~` retourne soit `nil` (si aucune correspondance n'est détectée) soit l'indice du premier caractère de la chaîne pertinente si une correspondance est trouvée :

1. Si vous faites partie des gens qui ont réussi à éviter d'apprendre les expressions régulières, je me permets de vous inciter à prendre le temps d'explorer cet outil extrêmement pratique. Vous pouvez commencer avec des livres mentionnés à l'annexe B, Aller plus loin.

```
/old/ =~ 'this old house' # 5 - l'indice de 'old'
/Russ Russell/ =~ 'Fred'  # nil - Fred n'est pas Russ ni Russell
/.*/ =~ 'any old string'  # 0 - cette expression correspond
                          # à toute chaîne de caractères
```

Il existe aussi `!~`, cet opérateur permet de vérifier que l'expression régulière ne correspond *pas* à une chaîne de caractères donnée.

Votre propre classe

Ruby ne serait pas un langage orienté objet si vous ne pouviez pas créer vos propres classes :

```
class BankAccount
  def initialize( account_owner )
    @owner = account_owner
    @balance = 0
  end

  def deposit( amount )
    @balance = @balance + amount
  end

  def withdraw( amount )
    @balance = @balance - amount
  end
end
```

Évidemment, la syntaxe de définition des classes en Ruby est aussi dépouillée et laconique que le reste du langage. La définition d'une classe commence par le mot clé `class` suivi par le nom de la classe :

```
class BankAccount
```

Souvenez-vous qu'en Ruby les noms de constantes commencent toujours par une lettre majuscule. Selon la philosophie Ruby, le nom d'une classe est donc une constante. Cela paraît logique, car le nom d'une classe fait toujours référence à la même chose, la classe. Par conséquent, tous les noms de classes en Ruby doivent commencer par une lettre majuscule, ce qui explique le fait que notre classe s'appelle `BankAccount` avec un "B" majuscule. La seule règle absolue est que le nom de classe doit commencer par une lettre majuscule, mais il faut noter que les programmeurs Ruby nomment typiquement leurs classes en utilisant la casse mixte comme en Java.

La première méthode de notre classe `BankAccount` est la méthode `initialize` :

```
def initialize( account_owner )
  @owner = account_owner
  @balance = 0
end
```

La méthode `initialize` est à la fois ordinaire et particulière. Elle est ordinaire dans la façon dont elle est construite : la ligne de déclaration de méthode débute avec le mot clé `def` suivi du nom de la méthode et de la liste des arguments si elle est présente. Notre méthode `initialize` prend ici un seul argument, `account_owner`.

Ensuite vient le corps de la méthode ; dans ce cas particulier, ce sont deux instructions d'affectation. La première opération de notre méthode récupère la valeur passée dans l'argument `account_owner` et l'affecte à l'étrange variable `@owner` :

```
@owner = account_owner
```

Les noms commençant par `@` dénotent des *variables d'instance.* Chaque instance de la classe `BankAccount` emportera avec elle sa propre copie de `@owner`. De même, chaque instance de la classe `BankAccount` emportera sa propre copie de `@balance` initialisée à zéro. Comme d'habitude, il n'y a pas de déclaration des variables `@owner` et `@balance`, nous inventons leurs noms *illico presto*.

Malgré le fait qu'`initialize` soit définie de la même façon que toutes les autres méthodes, elle est particulière du fait de son nom. Ruby utilise la méthode `initialize` pour construire des nouveaux objets. Lorsque Ruby crée une nouvelle instance d'une classe, la méthode `initialize` est appelée afin de préparer l'objet avant utilisation. Si l'on ne définit pas la méthode `initialize` dans sa classe, Ruby suivra la logique orientée objet : il remontera dans la hiérarchie de classes jusqu'à ce qu'il trouve la méthode `initialize` ou qu'il arrive à la classe `Object`. La classe `Object` définit la méthode `initialize` (qui ne fait rien), ce qui garantit que la recherche s'achèvera à ce niveau. En substance, `initialize` est un constructeur à la façon Ruby.

Pour créer une nouvelle instance de notre classe, nous appelons la méthode `new`. Cette méthode doit utiliser les mêmes paramètres que la méthode `initialize` :

```
my_account = BankAccount.new('Russ')
```

Cette instruction alloue un nouvel objet `BankAccount`, appelle sa méthode `initialize` avec les arguments passés par `new` et affecte cette nouvelle instance de `BankAccount` à la variable `my_account`.

Notre classe `BankAccount` dispose de deux autres méthodes, `deposit` et `withdraw`, qui augmentent et réduisent respectivement le solde du compte. Mais comment accéder à la position du compte ?

Accès aux variables d'instance

Notre classe `BankAccount` semble prête à être utilisée, mais il reste un problème : en Ruby les variables d'instance d'un objet ne sont pas accessibles de l'extérieur. Si on créait un objet `BankAccount` et qu'on essayait d'obtenir la valeur de `@balance`, on aurait une surprise désagréable. L'exécution de ce code

```
my_account = BankAccount.new ('russ')
puts(my_account.balance)
```

provoque l'erreur

```
account.rb:8: undefined method 'balance' ... (NoMethodError)
```

De même, `my_account.@balance` ne fonctionne pas non plus. Les variables d'instance des objets Ruby ne sont tout simplement pas accessibles de l'extérieur. Alors, que faire ? Nous pouvons définir un accesseur :

```
def balance
  @balance
end
```

Il faut noter qu'il manque une instruction de retour à notre méthode `balance`. En l'absence d'une instruction de retour explicite, une méthode retourne la valeur de la dernière expression évaluée, ce qui correspond dans notre cas à `@balance`.

Notre `balance` est désormais accessible :

```
puts(my_account.balance)
```

Le fait d'omettre les parenthèses autour de la liste des arguments vide nous donne l'agréable impression d'accéder à la valeur au lieu d'appeler une méthode.

On pourrait avoir besoin de modifier la valeur de balance. La solution évidente serait d'ajouter à `BankAccount` un mutateur :

```
def set_balance(new_balance)
  @balance = new_balance
end
```

Le code ayant une référence vers une instance de `BankAccount` pourrait ensuite modifier la valeur de `balance` :

```
my_account.set_balance(100)
```

Le problème, avec la méthode `set_balance`, c'est sa laideur. Ce serait beaucoup plus clair si l'on pouvait écrire :

```
my_account.balance = 100
```

Heureusement, c'est possible. Lorsque Ruby reçoit une telle expression d'affectation, il la traduira en un bon vieil appel de méthode. Le nom de la méthode sera le nom de la variable suivi par le signe égal. La méthode aura un seul paramètre : la valeur de la partie droite de l'instruction d'affectation. L'affectation ci-dessus sera donc traduite en l'appel de méthode suivant :

```
my_account.balance=(100)
```

Regardez bien le nom de cette méthode. Non, ce n'est pas une syntaxe spécifique, le nom de la méthode finit vraiment par le signe égal. Pour que tout cela fonctionne avec notre objet `BankAccount`, il suffit de renommer le mutateur :

```
def balance=(new_balance)
  @balance = new_balance
end
```

Maintenant, notre classe a bonne mine vue de l'extérieur : le code utilisant `BankAccount` peut affecter et récupérer la valeur de `balance` à volonté, sans se préoccuper du fait qu'en réalité il appelle les méthodes `balance` et `balance=`. Malheureusement, à l'intérieur notre classe est légèrement verbeuse. Il semble que nous soyons condamnés à voir toutes ces méthodes ennuyeuses encombrer notre définition de classe. Il n'en est rien. Ruby va à nouveau venir à notre secours.

Il s'avère que les accesseurs et mutateurs sont tellement fréquents que Ruby fournit un raccourci formidable pour les créer. Au lieu d'écrire chaque `def name`, etc., nous pouvons simplement ajouter une ligne à notre classe :

```
attr_accessor :balance
```

Cette expression crée une méthode qui s'appelle `balance` et qui ne fait rien d'autre que retourner la valeur de `@balance`. Elle crée également le mutateur `balance=(new_value)`. Il est même possible de créer des accesseurs multiples en une instruction :

```
attr_accessor :balance, :grace, :agility
```

Le code ci-dessus ajoute pas moins de six nouvelles méthodes à sa classe d'appartenance : ce sont les accesseurs et mutateurs pour chacune des trois variables d'instance[1]. Des accesseurs en un clin d'œil.

---

1. Il existe ici une subtilité de terminologie : les variables commençant par le signe arobase à l'intérieur de la classe sont des variables d'instance. Lorsque vous créez des accesseurs et des mutateurs, elles deviennent des attributs de l'objet, d'où les noms `attr_reader` et `attr_writer`. En pratique, ces points de détail de la terminologie ne désorientent personne, donc on n'y prête pas beaucoup d'attention.

De la même manière, si le monde extérieur doit pouvoir lire vos variables d'instance mais pas les modifier, il vous suffit de remplacer `attr_accessor` par `attr_reader` :

```
attr_reader :name
```

Désormais, votre classe expose un accesseur, mais pas de mutateur. De même, `attr_writer` crée seulement le mutateur, `name= (new_value)`.

Un objet demande : qui suis-je ?

Parfois, une méthode a besoin d'obtenir une référence vers l'objet courant, l'instance à laquelle la méthode appartient. Dans ce cas, on peut utiliser `self`, qui est toujours la référence vers l'objet courant :

```
class SelfCentered
  def talk_about_me
    puts("Hello I am #{self}")
  end
end
conceited = SelfCentered.new
conceited.talk_about_me
```

Le résultat de l'exécution de ce code ressemble à

```
Hello I am #<SelfCentered:0x40228348>
```

Évidemment, il est peu probable que votre instance de `SelfCentered` réside à la même adresse hexadécimale que la mienne, donc, votre résultat sera légèrement différent.

Héritage, classes filles et classes mères

Ruby supporte l'héritage simple, toutes les classes que l'on crée ne peuvent avoir qu'un seul parent ou classe-mère. Si la classe parent n'est pas spécifiée, votre classe devient automatiquement une classe fille de `Object`. Si vous voulez choisir une classe mère autre que `Object`, vous devez le spécifier après le nom de la classe :

```
class InterestBearingAccount < BankAccount
  def initialize(owner, rate)
    @owner = owner
    @balance = 0
    @rate = rate
  end

  def deposit_interest
    @balance += @rate * @balance
  end
end
```

Regardez bien la méthode `initialize` de `InterestBearingAccount`. Tout comme la méthode `initialize` de `BankAccount`, elle remplit les variables d'instance `@owner` et `@balance` ainsi que la nouvelle variable d'instance `@rate`. Le point clé est la correspondance des variables d'instance `@owner` et `@balance` de `InterestBearingAccount` avec celles de la classe `BankAccount`. En Ruby, une instance d'objet ne possède qu'un seul ensemble de variables d'instance visibles à travers tout son arbre hiérarchique. Si sur un coup de folie on décidait de créer une classe fille de `BankAccount` et puis sa classe fille, etc. jusqu'à faire quarante classes et quarante classes filles, il n'y aurait qu'une seule instance de `@owner` par instance de classe.

Un aspect regrettable de notre classe `InterestBearingAccount` est le fait que sa méthode `initialize` affecte des valeurs à des champs `@owner` et `@balance`, créant un doublon de la méthode `initialize` de `BankAccount`. Nous pouvons rendre le code plus propre en appelant la méthode `initialize` de la classe `Account` à partir de la méthode `initialize` de `InterestBearingAccount` :

```
def initialize(owner, rate)
  super(owner)
  @rate = rate
end
```

Notre nouvelle méthode `initialize` remplace le code dupliqué par l'appel à `super`. Lorsqu'une méthode appelle `super`, elle cherche à appeler une méthode avec le même nom dans sa classe mère. L'effet de l'appel à `super` dans la méthode `initialize` est d'appeler la méthode `initialize` de la classe `BankAccount`. Si la méthode avec le même nom n'existe pas dans la classe mère, Ruby continue à remonter l'arbre de l'héritage jusqu'à ce qu'il trouve la méthode ou arrive à la fin de la hiérarchie. Dans le deuxième cas, une erreur surviendra.

Contrairement à beaucoup de langages orientés objet, Ruby n'assure pas automatiquement l'appel des méthodes `initialize` dans toutes vos classes mères. Dans ce sens, Ruby considère `initialize` comme une méthode ordinaire. Si la méthode `initialize` de `InterestBearingAccount` n'appelait pas `super`, la version d'`initialize` dans `BankAccount` ne serait jamais appelée par `InterestBearingAccount`.

Options pour les listes d'arguments

Jusqu'ici, les méthodes que nous avons créées pour décorer nos classes fournissaient des listes d'arguments assez ordinaires. Il s'avère que Ruby fournit un grand nombre d'options pour les arguments de méthode. Par exemple, il est possible de spécifier des valeurs par défaut pour nos arguments :

```
def create_car( model, convertible=false)
  # ...
end
```

On peut appeler `create_car` soit avec deux arguments soit avec un seul, dans le dernier cas la variable `convertible` reçoit la valeur `false` par défaut. Toutes les expressions suivantes sont des idiomes valides pour créer une classe `Car` :

```
create_car('sedan')
create_car('sports car', true)
create_car('minivan', false)
```

Si vous écrivez des méthodes comme dans l'exemple précédent, les arguments qui acceptent des valeurs par défaut doivent se trouver à la fin de la liste d'arguments.

La possibilité de définir des valeurs par défaut donne beaucoup de flexibilité mais, parfois, il est commode d'avoir encore plus de liberté. Dans ce cas, on peut créer des méthodes acceptant un nombre arbitraire d'arguments :

```
def add_students(*names)
  for student in names
    puts("adding student #{student}")
  end
end
add_students( "Fred Smith", "Bob Tanner" )
```

Exécutez le code ci-dessus et vous obtiendrez le résultat suivant :

```
adding student Fred Smith
adding student Bob Tanner
```

La méthode `add_students` fonctionne car les arguments sont emballés dans un tableau, c'est ce qu'indique l'astérisque. On peut même mélanger les arguments ordinaires avec les tableaux d'arguments, du moment que le tableau est placé à la fin de la liste :

```
def describe_hero(name, *super_powers)
  puts("Name: #{name}")
  for power in super_powers
    puts("Super power: #{power}")
  end
end
```

La méthode précédente requiert au moins un argument, mais elle acceptera autant d'arguments que vous le souhaitez. Tous les appels ci-après sont donc valides :

```
describe_hero("Batman")
describe_hero("Flash", "speed")
describe_hero("Superman" , "can fly", "x-ray vision", "invulnerable")
```

Modules

En complément des classes, Ruby fournit un deuxième moyen d'encapsuler du code : les modules. Tout comme une classe, un module est un lot de méthodes et de constantes. Contrairement à une classe, on ne peut jamais créer une instance d'un module. Néanmoins, un module peut être inclus dans une classe qui récupérera ainsi les méthodes et constantes du module pour les rendre disponibles aux instances de la classe. Si vous êtes un programmeur Java, vous pouvez considérer les modules comme des interfaces qui contiennent des fragments du code d'implémentation.

Une définition de module ressemble singulièrement à une définition de classe. Voici l'exemple d'un module comprenant une méthode :

```
module HelloModule
  def say_hello
    puts('Hello out there.')
  end
end
```

Une fois la méthode définie, nous pouvons l'importer dans nos classes avec l'instruction `include`[1] :

```
class TryIt
  include HelloModule
end
```

L'instruction `include` a pour effet de rendre toutes les méthodes de module disponibles pour les instances de classe :

```
tryit = TryIt.new
tryit.say_hello
```

L'accessibilité est réciproque : une fois le module inclus dans la classe, toutes les méthodes et les variables d'instance de la classe deviennent disponibles pour les méthodes du module. Par exemple, le module suivant comprend une méthode qui affiche différentes informations concernant l'objet dans lequel le module se trouve. Il récupère ces valeurs en appelant les méthodes `name`, `title` et `department` exposées par la classe hôte :

```
module Chatty
  def say_hi
    puts("Hello, my name is #{name}")
    puts("My job title is #{title}")
    puts("I work in the #{department} department")
  end
end
```

1. Vous découvrirez un autre moyen d'utiliser les modules au Chapitre 12 : on peut simplement appeler leurs méthodes directement, sans les inclure dans des classes.

```
class Employee
  include Chatty
  def name
    'Fred'
  end

  def title
    'Janitor'
  end

  def department
    'Maintenance'
  end
end
```

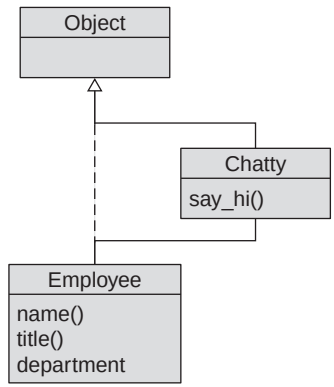
Le résultat de l'exécution est :

```
Hello, my name is Fred
My job title is Janitor
I work in the Maintenance department
```

Lorsque le module est inclus dans votre classe, il devient une sorte de classe secrète et particulière de sa classe mère (voir Figure 2.1). Mais, bien qu'une classe ne puisse avoir qu'une seule classe mère, elle peut inclure un nombre illimité de modules.

Figure 2.1

Un module importé dans une classe



Lorsqu'une méthode est appelée sur une instance de classe, Ruby vérifie tout d'abord si cette méthode est définie dans cette classe même. Si c'est le cas, la méthode est appelée. Par exemple, lorsque vous appelez la méthode `name` sur une instance de `Employee`, Ruby commence par chercher si la méthode est disponible dans la classe `Employee` et fait appel à elle. Si la méthode n'est pas disponible directement dans la classe, Ruby continue sa recherche au sein des modules inclus dans la classe. Par exemple, si vous appelez la méthode `say_hi`, Ruby, s'il ne la trouve pas dans la classe `Employee`, continuera de chercher dans les modules inclus par `Employee`. Si plusieurs modules sont

inclus dans la classe, Ruby effectue la recherche dans les modules en commençant par le dernier module inclus. Ici, notre classe `Employee` n'inclut qu'un module ; Ruby trouvera et appellera la méthode `say_hi` du module `Chatty`. Si Ruby ne trouvait pas la méthode dans la classe `Employee` ou dans l'un de ses modules, il continuerait la recherche dans la classe mère de la classe `Employee` puis dans ses modules.

Les modules utilisés de la façon décrite ci-dessus sont appelés des mixins, ils sont là pour être "mixés" ou mélangés avec des classes pour leur adjoindre leurs méthodes.

Conceptuellement, les mixins ressemblent aux interfaces Java et C#. Tout comme une interface, un module permet à des classes similaires de partager un ensemble de méthodes communes. La différence réside dans le fait qu'une interface reste complètement abstraite – elle ne fournit aucune implémentation – tandis qu'un module est fourni complet, avec l'implémentation.

Exceptions

La plupart des langages actuels ont une facilité pour gérer les malheurs de l'informatique, qui parfois affectent même le code le plus respectable. Ruby n'est pas une exception. Lorsqu'un ennui survient dans un programme, l'interpréteur Ruby arrête l'exécution et lève une exception. L'exception remonte la pile d'appels comme une bulle d'air jusqu'à ce que Ruby rencontre du code chargé de gérer l'exception ou jusqu'à ce qu'il arrive à la fin de la pile. Dans le dernier cas, Ruby termine votre programme. On peut attraper les exceptions à l'aide de la paire d'instructions `begin/rescue` :

```
begin
  quotient =1/0 # Boom!
rescue
  puts('Something bad happened')
end
```

Ruby intercepte toute exception pouvant se produire entre les instructions `begin` et `rescue` et rend immédiatement la main au code qui se trouve après l'instruction `rescue`. Vous pouvez préciser les types d'erreurs que vous voulez gérer en ajoutant la liste des classes d'exception dans l'instruction `rescue` :

```
begin
  quotient =1/0 # Boom!
rescue ZeroDivisionError
  puts('You tried to divide by zero')
end
```

Si vous vous retrouvez à la source du problème plutôt que dans le rôle d'une victime, vous pouvez lancer votre propre exception avec l'instruction `raise` :

```
if denominator == 0
  raise ZeroDivisionError
end
return numerator / denominator
```

Ruby fournit plusieurs raccourcis bien pratiques pour lancer des exceptions. Si votre instruction `raise` appelle une classe d'exception, comme nous l'avons fait dans l'exemple précédent, Ruby crée commodément une nouvelle instance de cette classe et l'utilise en tant qu'exception. Inversement, lorsque vous définissez une instruction `raise` avec une chaîne de caractères, Ruby instancie la classe `RuntimeException` et utilise la chaîne comme message incorporé dans cette exception :

```
irb(main):001:0> raise 'You did it wrong'
RuntimeError: You did it wrong
```

Threads

À l'instar de nombreux langages récents, Ruby a un système incorporé de fils d'exécution ou threads. Les threads permettent à vos programmes de faire plusieurs choses à la fois[1]. Créer des threads en Ruby est simple : le constructeur de la classe `Thread` accepte un bloc qui devient le corps du thread. L'exécution du thread commence au moment de sa création et continue jusqu'à la fin du bloc. Voici un exemple de deux threads qui calculent la somme et le produit des dix premiers entiers :

```
thread1 = Thread.new do
  sum=0
  1.upto(10) {|x| sum = sum + x}
  puts("The sum of the first 10 integers is #{sum}")
end

thread2 = Thread.new do
  product=1
  1.upto(10) {|x| product = product * x}
  puts("The product of the first 10 integers is #{product}")
end

thread1.join
thread2.join
```

1. Si vous travaillez sur un système monoprocesseur, les threads provoquent seulement l'illusion de faire plusieurs choses à la fois. En réalité, le système avance légèrement sur une tâche avant de passer la main à la tâche suivante. Mais les choses se passent si rapidement qu'il est la plupart du temps impossible de s'apercevoir de la différence.

Vous pouvez attendre la fin de l'exécution d'un thread en utilisant la méthode `join` :

```
thread1.join
thread2.join
```

Bien que le code multitâche puisse être très puissant, il est aussi très dangereux. Permettre à deux threads ou plus de modifier la même structure de données simultanément est en général un moyen redoutable d'introduire des bugs très difficiles à trouver. Une bonne solution pour éviter cela, ainsi que d'autres situations de concurrence, et protéger votre code contre l'accès simultané est d'utiliser la classe `Monitor` :

```
@monitor = Monitor.new
@monitor.synchronize do
  # Accédé par un thread à la fois...
end
```

Gérer des fichiers de code source séparés

Les exemples de programmation que nous avons utilisés présentent cet avantage d'être suffisamment courts pour être logés dans un seul fichier source. Malheureusement, la plupart des applications réelles deviennent trop grandes pour un seul fichier. La réponse logique consiste à diviser le système en plusieurs fichiers contenant des fragments de code gérables. Dès que votre système est divisé en plusieurs fichiers, vous vous retrouvez face au problème de chargement de tous ces fichiers. Selon le langage utilisé, ce problème est résolu de façons différentes. Java, par exemple, possède un système complexe de chargement automatique de classes lorsque le programme en a besoin.

L'approche Ruby est différente. Les programmes Ruby doivent explicitement charger les fichiers dont ils dépendent. Par exemple, si votre classe `BankAccount` est définie dans `account.rb` et qu'elle doit être utilisée par la classe `Portfolio`, qui réside dans `portfolio.rb`, vous devez vous assurer que `BankAccount` est chargé avant que `Portfolio` ne commence à l'utiliser. Cette tâche est accomplie à l'aide de l'instruction `require` :

```
require 'account.rb'
class Portfolio
  # Uses BankAccount
end
```

L'instruction `require` charge le contenu d'un fichier dans l'interpréteur Ruby. Cette instruction est assez intelligente : elle ajoute le suffixe `.rb` automatiquement. Les programmeurs Ruby écriront donc plus simplement :

```
require 'account'
```

L'instruction `require` retient également les noms des fichiers qui sont déjà chargés et ne charge pas le même fichier deux fois. Si le chargement d'un fichier est demandé plusieurs fois dans votre code, cela ne pose aucun problème. Puisque `require` gère si bien les fichiers à charger, les programmeurs demandent habituellement le chargement de tous les fichiers nécessaires au début de chaque fichier Ruby sans s'inquiéter des classes qui ont déjà été chargées par un autre fichier.

Tout ceci s'applique non seulement aux fichiers que vous produisez, mais aussi aux fichiers livrés avec la bibliothèque standard de Ruby. Par exemple, si vous devez analyser des URL, vous pouvez simplement charger la classe URI fournie avec Ruby :

```
require 'uri'
yahoo = URI.parse('http://www.yahoo.com')
```

Une dernière précision sur la saga `require` concerne les gemmes RubyGems. RubyGems est un système de paquetage qui permet aux développeurs de publier des bibliothèques et applications Ruby sous la forme de paquets pratiques et faciles à installer. Si vous voulez utiliser une bibliothèque provenant d'une gemme, par exemple la gemme nommée `runt`[1], il faut commencer par inclure RubyGems :

```
require 'rubygems'
require 'runt'
```

En conclusion

De `hello_world` aux modules et à `require`, ce chapitre était un cours de Ruby express. Fort heureusement, de nombreux éléments de base de Ruby – les nombres, les chaînes de caractères et les variables – sont relativement standard. Les spécificités du langage telles que les constantes qui ne sont pas très constantes et le fait que zéro soit `true` ne sont pas bouleversantes. À ce stade, après avoir jeté un œil sur les bases du langage, nous arrivons déjà à avoir un aperçu des raisons qui font de Ruby un langage très agréable. La syntaxe est concise, mais pas cryptique. Tout ce que l'on trouve dans un programme – de la chaîne `'abc'` au nombre `42` en passant par des tableaux – est objet.

Dans les chapitres qui suivent, nous verrons des design patterns (motifs de conception) et nous comprendrons comment Ruby nous donne la possibilité d'exprimer des choses extrêmement puissantes de manière claire et laconique.

---

1. Au Chapitre 15, nous en apprendrons plus sur `runt`, qui est une bibliothèque consacrée à la gestion du temps et des tâches planifiées.

Partie II

Patterns en Ruby

3

Varier un algorithme avec le pattern Template Method

Imaginez que vous ayez un fragment de code complexe : un algorithme compliqué, un enchevêtrement de code métier ou un fragment suffisamment difficile pour vouloir l'écrire une seule fois, rajouter des tests unitaires et le laisser en paix. Le problème est qu'en plein milieu de votre code complexe se trouve une partie qui doit être variable. Parfois, elle est exécutée d'une façon et parfois d'une autre. Pire encore, vous êtes presque certain que dans le futur cette partie devra prendre des responsabilités supplémentaires. Vous êtes face au vieux problème "comment se protéger contre les changements" que nous avons examiné au Chapitre 1. Que faire ?

Pour concrétiser davantage le scénario, imaginez que votre premier vrai projet Ruby consiste à écrire un générateur de rapports, c'est-à-dire un programme qui produira des rapports d'avancement mensuels. Les rapports doivent être joliment mis en forme en HTML et vous écrivez donc quelque chose comme ceci :

```
class Report
  def initialize
    @title = 'Monthly Report'
    @text = [ 'Things are going', 'really, really well.' ]
  end

  def output_report
    puts('<html>')
    puts(' <head>')
    puts(" <title>#{@title}</title>")
    puts(' </head>')
    puts(' <body>')
    @text.each do |line
      puts(" <p>#{line}</p>" )
    end
```

```
      puts(' </body>') puts('</html>')
    end
  end
```

Il est clair que nous avons pris quelques libertés avec ce code pour lui conserver une certaine simplicité. Dans la vraie vie, le rapport ne serait pas codé en dur dans la classe, et nous n'insérerions pas du texte arbitraire dans un fichier HTML sans vérifier la présence de balises "<" et ">". Ceci dit, le code précédent présente quelques bonnes caractéristiques. Il est simple, facile à utiliser et il produit effectivement du HTML :

```
report = Report.new
report.output_report
```

Si vous devez juste générer du HTML basique, ce code, ou du code semblable, est tout ce dont vous avez besoin.

Faire face aux défis de la vie

Malheureusement, même si les choses sont simples au début, elles le demeurent rarement. Quelques mois après avoir fini le chef-d'œuvre de programmation précédent, vous recevez une nouvelle demande : l'objet qui gère le formatage doit produire du texte simple en plus du HTML. Et avant la fin de l'année on aura probablement besoin d'un format PostScript et peut-être RTF.

Parfois, les solutions les plus simples sont les meilleures, donc vous abordez le problème de la façon la plus bête possible :

```
class Report
  def initialize
    @title = 'Monthly Report'
    @text = ['Things are going', 'really, really well.']
  end

  def output_report(format)
    if format == :plain
      puts("*** #{@title} ***")
    elsif format == :html
      puts ( ' <html> ' )
      puts(' <head>')
      puts(" <title>#{@title}</title>")
      puts(' </head>')
      puts(' <body>')
    else
      raise "Unknown format: #{format}"
    end
    @text.each do |line|
      if format == :plain
        puts(line)
      else
```

```
          puts(" <p>#{line}</p>" )
        end
      end
      if format == :html
        puts(' </body>')
        puts('</html>')
      end
    end
  end
```

Beurk ! Cette deuxième version fonctionne assurément, mais c'est un véritable bazar. Le code traitant du texte simple est mélangé avec le code qui concerne le HTML. Pire encore, lorsque vous ajouterez de nouveaux formats (il ne faut pas oublier la demande pour PostScript qui vient de surgir), vous serez obligé de retravailler la classe `Report` pour intégrer chaque nouveauté. Vu la façon dont le code est construit actuellement, avec l'ajout de chaque nouveau format vous courez le risque de perturber le bon fonctionnement des formats existants. Bref, notre première tentative d'ajouter un nouveau format de sortie est en violation d'un des principes fondamentaux des design patterns : elle mélange le code variable avec le code permanent.

Séparer les choses qui restent identiques

Un moyen de sortir de l'embarras consiste à remanier ce désastre pour séparer le code qui gère les formats différents. La clé est de se rendre compte que, quel que soit le format – le texte simple, le HTML ou le futur PostScript –, le flux d'exécution de `Report` demeure le même :

1. Afficher l'en-tête en fonction du format spécifique.

2. Afficher le titre.

3. Afficher chaque ligne du compte rendu.

4. Afficher les balises fermantes requises par un format donné.

En gardant cette séquence à l'esprit, retournons aux premières leçons de la programmation orientée objet : définir une classe abstraite avec des méthodes pour gérer les étapes énumérées ci-dessus et laisser les détails de chaque étape aux classes filles. En utilisant cette approche nous nous retrouvons avec une classe fille qui correspond à chaque format de sortie. Voici notre nouvelle classe abstraite `Report` :

```
class Report
  def initialize
    @title = 'Monthly Report'
    @text = ['Things are going', 'really, really well.']
  end
```

```
  def output_report
    output_start
    output_head
    output_body_start
    output_body
    output_body_end
    output_end
  end

  def output_body
    @text.each do |line|
      output_line(line)
    end
  end

  def output_start
    raise 'Called abstract method: output_start'
  end

  def output_head
    raise 'Called abstract method: output_head'
  end

  def output_body_start
    raise 'Called abstract method: output_body_start'
  end

  def output_line(line)
    raise 'Called abstract method: output_line'
  end

  def output_body_end
    raise 'Called abstract method: output_body_end'
  end

  def output_end
    raise 'Called abstract method: output_end'
  end
end
```

Certes, la classe `Report` n'est pas complètement abstraite. Nous pouvons théoriser sur les méthodes et classes abstraites, mais en vérité Ruby ne supporte aucune des deux. L'idée des méthodes et des classes statiques n'est pas tout à fait compatible avec le mode de vie simple et dynamique de Ruby. Le mieux que l'on puisse faire est de déclencher des exceptions dans le cas où un utilisateur essaie d'appeler une de nos méthodes "abstraites".

Une fois notre nouvelle implémentation de `Report` terminée, nous pouvons définir des classes filles de `Report` pour chacun des deux formats. Voici la classe qui traite le HTML :

```
class HTMLReport < Report
  def output_start
    puts('<html>')
  end
```

```
    def output_head
      puts(' <head>')
      puts(" <title>#{@title}</title>")
      puts(' </head>')
    end

    def output_body_start
      puts('<body>')
    end

    def output_line(line)
      puts(" <p>#{line}</p>")
    end

    def output_body_end
      puts('</body>')
    end

    def output_end
      puts('</html>')
    end
  end
```

Et voici la version qui traite le texte simple :

```
  class PlainTextReport < Report
    def output_start
    end

    def output_head
      puts("**** #{@title} ****")
      puts
    end

    def output_body_start
    end

    def output_line(line)
      puts(line)
    end

    def output_body_end
    end

    def output_end
    end
  end
```

L'usage de nos nouvelles classes est simple :

```
  report = HTMLReport.new
  report.output_report
  report = PlainTextReport.new
  report.output_report
```

Choisir le format est facile, il suffit de sélectionner la classe de formatage correspondant.

Découvrir le pattern Template Method

Félicitations ! Vous venez de redécouvrir ce qui constitue probablement le plus simple des patterns répertoriés par le GoF, le pattern Template Method.

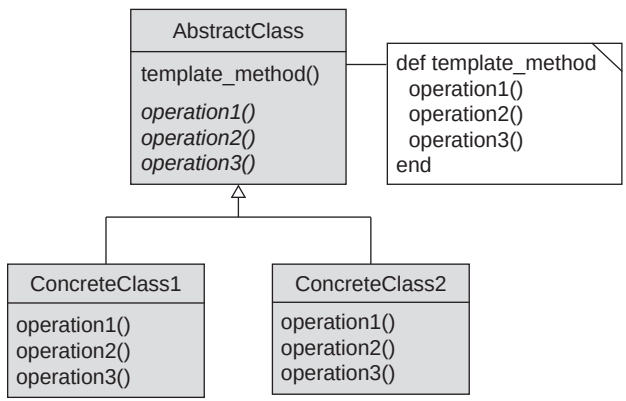
Comme vous le voyez à la Figure 3.1, l'idée générale du pattern Template Method consiste à construire une classe parent abstraite avec un squelette de méthode. Ce squelette (autrement appelé *template method*) actionne le traitement variable en faisant appel aux méthodes abstraites, dont l'implémentation est fournie par des classes filles. Les différences de traitement sont conditionnées par la sélection des classes filles concrètes.

Dans notre exemple, le plan de base comprend toutes les étapes nécessaires pour générer un rapport : afficher l'information de l'en-tête, le titre du rapport puis chaque ligne du texte. Dans ce cas, les implémentations de méthodes dans les classes filles se chargent d'écrire le rapport dans le bon format : soit du texte simple soit du HTML. Si l'on réussit à concevoir les tâches correctement, on arrivera à une séparation des parties statiques (l'algorithme principal exprimé dans template method) et des parties variables (les détails fournis par les classes filles).

Les classes `HTMLReport` et `PlainTextReport` paraissent incomplètes, c'est une caractéristique qu'elles partagent avec toutes les classes concrètes bien écrites qui suivent le pattern Template Method. En tant que bonnes classes concrètes, `HTMLReport` et `PlainTextReport` surchargent toutes les deux des méthodes abstraites telles que `output_line`. Leur apparence fragmentaire provient du fait qu'elles ne surchargent pas la méthode clé – template method – `output_report`. Selon le modèle Template Method, la classe abstraite contrôle le traitement de haut niveau à l'aide du pattern template method, les classes filles remplissant simplement les détails.

Figure 3.1

Diagramme de classes du pattern Template Method



Méthodes d'accrochage

Si vous relisez `PlainTextReport`, vous remarquerez que cette classe surcharge les méthodes `output_start` et `output_end` ainsi que les méthodes qui concernent le corps du rapport, mais ces méthodes ne comprennent pas de code d'implémentation. Ceci paraît raisonnable : contrairement au document HTML, un document en texte simple ne nécessite ni en-tête ni balise fermante. Cependant, il n'y a aucune raison de définir toutes ces méthodes dans une classe comme `PlainTextReport`, qui n'en a pas besoin. Il est plus judicieux de fournir une implémentation par défaut de ces méthodes dans la classe parent `Report` afin que les classes filles puissent y avoir accès :

```
class Report
  def initialize
    @title = 'Monthly Report'
    @text = ['Things are going', 'really, really well.']
  end

  def output_report
    output_start
    output_head
    output_body_start
    @text.each do |line|
      output_line(line)
    end
    output_body_end
    output_end
  end

  def output_start
  end

  def output_head
    raise 'Called abstract method: output_head'
  end

  def output_body_start
  end

  def output_line(line)
    raise 'Called abstract method: output_line'
  end

  def output_body_end
  end

  def output_end
  end
end
```

Les méthodes non abstraites qui peuvent être surchargées par des classes concrètes du pattern Template Method s'appellent des *hooks* (ou point d'accroche). Une méthode

d'accrochage (*hook*) permet à des classes concrètes soit de surcharger l'implémentation de base pour faire du traitement différencié, soit d'accepter l'implémentation par défaut. Souvent, les classes mères définissent des hooks dans le but de tenir les sous-classes concrètes au courant de ce qui se passe dans le code. Lorsque la classe `Report` appelle `output_start`, cela signifie pour les sous-classes que "nous sommes prêts pour afficher le rapport, si vous avez des choses à faire, c'est le moment de le faire". Les implémentations par défaut de ces hooks informatifs sont fréquemment vides. Leur raison d'être est d'informer les sous-classes de l'avancement de l'exécution sans les obliger à surcharger des méthodes qui n'ont pour eux aucun intérêt.

Parfois, l'implémentation par défaut d'un hook peut contenir du code. Dans notre exemple de la classe `Report`, nous avons une possibilité de gérer le titre comme s'il s'agissait d'une simple ligne de texte :

```
class Report
  def initialize
    @title = 'Monthly Report'
    @text = ['Things are going', 'really, really well.']
  end

  def output_report
    output_start
    output_head
    output_body_start
    @text.each do |line|
      output_line(line)
    end
    output_body_end
    output_end
  end

  def output_start
  end

  def output_head
    output_line(@title)
  end

  def output_body_start
  end

  def output_line(line)
    raise 'Called abstract method: output_line'
  end

  def output_body_end
  end

  def output_end
  end
end
```

Mais où sont passées toutes les déclarations ?

Vu que ce chapitre décrit notre premier patron de conception Ruby, cela vaut la peine de prendre un moment pour parler des types et de la sécurité de type en Ruby. Si vous arrivez du monde des langages à typage statique, vous vous demandez probablement comment notre classe `Report` et ses sous-classes peuvent tenir la route sans pratiquement effectuer aucune déclaration de type. Vous avez remarqué que nous ne définissons nulle part dans la classe `Report` que `@title` est une chaîne de caractères et `@text`, un tableau de chaînes de caractères. Suivant la même logique, lorsque le code client crée un nouvel `HTMLReport`, nous ne précisons jamais que la variable `report` contient une référence vers une instance de `HTMLReport` ou de `Report` ; nous écrivons simplement :

```
report = HTMLReport.new
```

Ruby est dynamiquement typé, ce qui signifie qu'aucune vérification n'est faite pour s'assurer qu'un objet particulier possède un ancêtre d'un type particulier. La seule chose qui importe, c'est que l'objet implémente les méthodes appelées par ses clients. Dans l'exemple précédent, la classe `Report` s'attend à ce que l'objet `@text` se comporte comme un tableau de chaînes de caractères. Si `@text` ressemble à un tableau de chaînes de caractères – à savoir que l'on peut en extraire sa troisième chaîne de caractères en appelant `@text[2]` –, le type est considéré comme correct quelle que soit sa classe réelle.

L'approche du typage "je suis ce que je fais" est connu sous le nom "*duck typing*" où "*typage à la canard*". Le nom provient d'un vieux principe qui consiste à dire que "si un objet ressemble à un canard et qu'il couine comme un canard, alors, ce doit être un canard". Une autre façon de voir le sujet est de considérer le fonctionnement du typage statique comme celui de l'aristocratie : les langages statiquement typés vous posent sans arrêt des questions sur votre parent ou grand-parent ou, dans le cas des interfaces à la Java, sur vos tantes et vos oncles. Dans les langages à typage statique, l'arbre de hiérarchie d'un objet a un rôle très important, tandis que les langages dynamiquement typés sont des méritocraties. Ils s'intéressent aux méthodes que possède la classe plutôt qu'à ses origines. Les langages dynamiquement typés ne se renseignent que rarement sur les ancêtres d'un objet. Ils disent plutôt : "Peu importe qui sont tes parents. Je veux seulement savoir ce que tu sais faire[1]."

Types, sécurité et flexibilité

Les gens qui sont habitués à programmer avec des langages statiquement typés se demandent souvent comment tout cela peut fonctionner. Vous avez probablement

1. À mon avis, ils le disent avec un accent d'un conducteur de taxi new-yorkais.

l'impression que le typage "à la canard", si libre et si simple, mènera inévitablement à une catastrophe, que les programmes s'arrêteront brutalement de fonctionner à force d'essayer de formater du HTML sur un objet de type connexion à une base de données ou bien de tenter de demander au nombre 42 de générer un rapport mensuel. Aussi étonnant que cela puisse paraître, des problèmes de typage aussi atroces n'arrivent que rarement.

Vous pouvez trouver des preuves de cette robustesse là où l'on n'aurait jamais pensé la trouver : dans le monde des programmes Java. La plupart des programmes Java écrits avant l'arrivée de Java 1.5 (ce qui couvre la majorité des programmes Java existants) utilisent des conteneurs du package `java.util` tels que `HashMap` et `ArrayList`. Les versions de ces conteneurs antérieures à Java 1.5 n'assuraient aucune sécurité de types, et même après l'arrivée de la version 1.5 Java continue à supporter les versions sans sécurité de type pour des raisons de compatibilité ascendante. Malgré cette attitude cavalière envers la sécurité des types, la plupart des programmes Java ne mélangent pas leurs objets socket avec des objets `Employee` et ne partent pas en vrille en essayant d'augmenter le salaire d'une connexion réseau.

Les langages à typage statique sont tellement omniprésents aujourd'hui que l'on se pose rarement la question clé : quel est le coût du typage statique ? Ma réponse est que le coût est très élevé. En terme d'effort de programmation et d'encombrement du code, le typage statique coûte une fortune. Regardez un programme en Java ou C# et comptez le nombre d'instructions consacrées à la déclaration de variables et paramètres. Rajoutez la majorité des définitions d'interfaces. N'oubliez pas les agaçantes instructions `cast`, lorsque vous essayez de convaincre le système que c'est vraiment un objet `String`. Rajoutez un bonus pour chaque déclaration générique complexe. Tout ce code très encombrant n'est pas gratuit[1].

Et il ne s'agit pas seulement d'effort de programmation. Bien qu'il soit caché, le coût du typage statique est bien réel : il remonte le couplage de votre système à un niveau bien plus élevé que nécessaire. Voyez la méthode Java `isEmpty()` :

```
public boolean isEmpty(String s) {
  return s.length() == 0;
}
```

1. Pour être impartial, il me faut préciser que le typage statique est très coûteux en terme d'encombrement de code à cause de l'implémentation des langages largement répandus aujourd'hui. Néanmoins, il existe un certain nombre de langages moins connus, tels qu'OCaml et Scala, qui gèrent le typage statique avec beaucoup moins de bruit.

Et maintenant son jumeau en Ruby :

```
def empty?(s)
  s.length == 0
end
```

Au premier abord, les méthodes sont équivalentes. Souvenez-vous que la version Java ne fonctionne qu'avec les arguments de type `java.lang.String`. La méthode Ruby est sûre de fonctionner avec des chaînes de caractères, mais elle fonctionnera tout aussi bien avec des tableaux, des queues, des collections et des tableaux associatifs. En fait, la méthode `empty?` en Ruby fonctionnera avec tout argument disposant de la méthode `length`. Elle se moque du type exact de l'argument, et c'est bien mieux ainsi.

Comparés au typage dynamique, ces arguments peuvent ne pas paraître intuitifs pour une personne habituée au typage statique. Si vous êtes habitué au typage statique et au fait de tout déclarer, la construction des grands systèmes fiables peut vous sembler irréaliste sans la vérification de type. Mais c'est possible et il y a deux exemples évidents pour démontrer cette possibilité.

Ruby on Rails est de loin la preuve la plus flagrante que l'écriture de code fiable est possible avec un langage dynamiquement typé. Rails est constitué de dizaines de milliers de lignes de code Ruby, et Rails est extrêmement stable. Si Rails ne vous convainc pas, pensez à un autre grand corps de code Ruby qui est utilisé tous les jours : la bibliothèque standard livrée avec Ruby. Cette bibliothèque compte plus de 100 000 lignes écrites en Ruby. Il est presque impossible d'écrire un programme en Ruby sans faire appel à cette bibliothèque, et elle fonctionne.

L'existence de Ruby on Rails et de la bibliothèque standard Ruby est la preuve qu'il est possible d'écrire des grands volumes de code fiable avec un langage dynamiquement typé. La crainte que le typage dynamique engendre un code chaotique est largement infondée. Il arrive effectivement que des programmes Ruby s'arrêtent occasionnellement à cause des problèmes de types, mais la fréquence de ces échecs est sans rapport avec les efforts déployés par les langages statiquement typés pour éviter une possibilité d'erreur aussi faible.

Est-ce que cela signifie que les langages dynamiquement typés sont meilleurs et que nous devrions complètement abandonner les langages à typage statique ? Le débat sur la question continue encore. Tout comme pour les autres questions relatives au génie logiciel, il existe deux côtés à la médaille. Le typage statique vaut la peine dans des grands systèmes complexes construits par des équipes immenses. Mais le typage dynamique présente un nombre d'avantages significatifs : la taille des programmes en Ruby est bien moindre par rapport à la taille de ses équivalents statiquement typés. Comme

nous l'avons vu dans l'exemple de la méthode `empty?` et le verrons encore dans les chapitres à venir, le typage dynamique offre une énorme flexibilité.

Si cela vous paraît fou, laissez-moi vous guider dans la suite de ce livre et donnez sa chance à cette folie dynamiquement typée. Il n'est pas impossible que vous soyez agréablement surpris.

Les tests unitaires ne sont pas facultatifs

Une des façons d'accroître vos chances d'être agréablement surpris consiste à écrire des tests unitaires. Quel que soit votre langage de travail, Java, C# ou Ruby, vous devez écrire des tests unitaires. La deuxième blague la plus ancienne[1] de la programmation est "ça se compile donc ça doit fonctionner".

Effectuer des tests est primordial quel que soit le langage, mais dans les langages dynamiques tels que Ruby ils sont critiques. Ruby ne dispose pas d'un compilateur pour effectuer une première passe de vérification et vous donner une fausse impression de sécurité. Le seul moyen de savoir si le programme fonctionne, c'est d'exécuter les tests. La bonne nouvelle est que les mêmes tests pourront à la fois démontrer que votre code fonctionne et vous aider à éliminer les éventuels problèmes de typage.

Une autre bonne nouvelle : si vous savez utiliser JUnit, NUnit ou une autre bibliothèque XUnit, alors, vous savez déjà écrire des tests unitaires en Ruby. Par exemple, la classe suivante vérifie notre méthode `empty?` :

```
require 'test/unit'
require 'empty'
class EmptyTest < Test::Unit::TestCase
  def setup
    @empty_string = ''
    @one_char_string = 'X'
    @long_string = 'The rain in Spain'
    @empty_array = []
    @one_element_array = [1]
    @long_array = [1, 2, 3, 4, 5, 6]
  end

  def test_empty_on_strings
    assert empty?(@empty_string)
    assert ! empty?(@one_char_string)
    assert ! empty?(@long_string)
  end
```

1. La blague la plus ancienne est le papillon de nuit mort, collé dans le journal d'erreurs.

```
  def test_empty_on_arrays
    assert empty?(@empty_array)
    assert ! empty?(@one_element_array)
    assert ! empty?(@long_array)
  end
end
```

`Test::Unit`, fidèle à ses racines XUnit, exécute chaque méthode dont le nom commence par `test` en tant que test. Si votre classe de test a une méthode `setup` (c'est le cas de la classe précédente), cette méthode est exécutée avant *chaque* méthode de test. Si votre classe possède une méthode `teardown` (la classe précédente ne l'a pas), elle est exécutée après *chaque* méthode de test.

`Test::Unit` est équipé de toute une collection de méthodes d'assertion. Vous pouvez vérifier qu'une valeur est à `true`, ou appeler `assert_equal` pour s'assurer de l'égalité de deux objets. Si vous voulez être sûr que l'objet n'est pas vide, utilisez `assert_not_nil`.

L'exécution des tests unitaires est très facile. Si le test ci-dessus est placé dans le fichier `string_test.rb`, on peut le démarrer en exécutant le fichier comme un programme Ruby :

```
$ ruby empty_test.rb
Loaded suite empty_test
Started
..
Finished in 0.000708 seconds.
2 tests, 6 assertions, 0 failures, 0 errors
```

Il est gratifiant (et terriblement sécurisant) de voir un test se terminer sans se plaindre.

User et abuser du pattern Template Method

Comme il est possible d'écrire une implémentation fiable du pattern Template Method en Ruby, reste à savoir comment nous y prendre. Procéder par itérations est la meilleure tactique : commencez par une variante du code et continuez à coder comme si vous aviez un seul problème à résoudre. Dans notre exemple, on pourrait commencer par le HTML :

```
class Report
  def initialize
    @title = 'MonthlyReport'
    @text = ['Things are going', 'really, really well.']
  end
```

```
    def output_report
      puts('<html>')
      puts(' <head>')
      puts(' <title>#{@title}</title>')
      # output the rest of the report ...
    end
  end
```

On pourrait ensuite refactoriser la méthode qui deviendra la méthode template afin qu'elle fasse appel à d'autres méthodes qui fourniront les parties variables de l'algorithme. Restez toujours focalisé sur un cas précis :

```
class Report
  # ...
  def output_report
    output_start
    output_title(@title)
    output_body_start
    for line in @text
      output_line(line)
    end
    output_body_end
    output_end
  end

  def output_start
    puts('<html>')
  end

  def output_title(title)
    puts(' <head>')
    puts(" <title>#{title}</title>")
    puts(' </head>')
  end
  # ...
end
```

Enfin, on pourrait créer une sous-classe pour le premier cas et placer toutes les implémentations spécifiques dedans. Maintenant, vous êtes prêt pour commencer à coder le reste des variations.

Comme mentionné au Chapitre 1, la pire erreur que vous puissiez faire est de pousser le bouchon trop loin en essayant de prévoir toutes les possibilités imaginables. Le pattern Template Method est optimal lorsqu'il est codé de façon concise – chaque méthode abstraite et chaque hook doivent avoir une raison d'exister. Évitez de créer une classe template qui oblige ses sous-classes à surcharger d'innombrables méthodes obscures dans une tentative désespérée de couvrir tous les cas de figure. Il est également déconseillé de créer une classe template incrustée d'une multitude de hooks que jamais personne ne surchargera.

Les Templates dans le monde réel

On peut trouver un exemple classique du pattern Template Method dans WEBrick, une bibliothèque légère, écrite complètement en Ruby, qui sert pour la création des serveurs TCP/IP. La partie clé de WEBrick est la classe `GenericServer`, qui implémente tous les détails de fonctionnement d'un serveur réseau. `GenericServer` n'a évidemment aucune idée de la façon dont vous voulez réaliser votre serveur. Donc, pour utiliser `GenericServer` on l'étend et on surcharge la méthode `run` :

```
require 'webrick'
class HelloServer < WEBrick::GenericServer
  def run(socket)
    socket.print('Hello TCP/IP world')
  end
end
```

La méthode template incorporée dans `GenericServer` contient tout le code pour écouter sur un port TCP/IP, accepter de nouvelles connexions et nettoyer le tout lorsque la connexion est interrompue. En plein milieu de tout ce code, précisément au moment où une nouvelle connexion est créée, votre méthode `run`[1] est appelée.

Il existe un autre exemple du pattern Template Method, qui est omniprésent au point d'en devenir difficilement visible. Pensez à la méthode `initialize`, que nous utilisons pour construire nos objets. Tout ce que nous savons, c'est que la méthode `initialize` est appelée vers la fin du processus de création d'objets et que nous pouvons la surcharger dans nos classes pour implémenter une initialisation spécifique. Cela ressemble fortement à une méthode hook.

En conclusion

Dans ce chapitre, nous avons étudié en détail notre premier design pattern, Template Method. Le pattern Template Method est simplement une façon prétentieuse de dire : si vous avez un algorithme variable, il peut être implémenté en déplaçant la partie statique dans une classe parent et en encapsulant les parties variables dans les méthodes définies par un nombre de classes filles. La classe parent peut laisser les méthodes sans implémentation, dans ce cas les sous-classes sont obligées de fournir l'implémentation.

---

1. Je sais tout cela parce que j'ai lu le code. Un des avantages des langages interprétés est le fait que le code source de la bibliothèque standard Ruby dort quelque part sur votre système en attendant de vous apprendre un tas de choses. Je ne peux pas imaginer une meilleure façon d'apprendre un nouveau langage de programmation que de lire du code qui fonctionne.

À l'inverse, la classe parent peut définir une implémentation par défaut des méthodes que les sous-classes peuvent choisir de surcharger ou pas.

Étant donné que c'est notre premier pattern, nous avons fait un détour pour explorer l'un des aspects les plus importants de la programmation en Ruby : le typage dynamique. Le typage à la canard est un compromis : vous abandonnez la sécurité de la vérification au moment de la compilation en échange d'une plus grande clarté du code et d'une flexibilité dans la programmation.

Nous verrons bientôt que le pattern Template Method est une brique de base de la programmation orientée objet, elle-même utilisée par d'autres patterns. Au Chapitre 13, par exemple, nous apprendrons que le pattern Factory Method n'est qu'un template method destiné à créer de nouveaux objets. Le problème auquel répond le pattern Template Method est également assez répandu. Au chapitre suivant, nous examinerons le pattern Strategy, qui propose une solution différente au même problème. Cette solution ne s'appuie pas sur l'héritage de la même manière que le pattern Template Method.

4

Remplacer un algorithme avec le pattern Strategy

Nous avons démarré le chapitre précédent avec la question : comment varier une partie de l'algorithme ? Comment permettre à l'étape 3 dans un processus de cinq étapes de faire parfois une chose et parfois une autre ? La réponse fournie au Chapitre 3 recourait au pattern Template Method : créer une classe parent avec une méthode Template, pour contrôler le traitement général, et laisser les sous-classes s'occuper des détails. Si aujourd'hui il nous faut un traitement particulier et demain un autre, nous définissons alors deux sous-classes : une pour chaque variation.

Malheureusement, le pattern Template Method présente quelques défauts dont la plupart proviennent de sa façon d'utiliser l'héritage. Comme nous l'avons vu au Chapitre 1, les modèles fondés sur l'héritage ont un certain nombre d'inconvénients importants. Quel que soit le soin que vous apportez à la conception de votre code, les sous-classes seront étroitement liées avec leurs classes parents, c'est la nature même de leur relation d'héritage. Qui plus est, les techniques reposant sur l'héritage, comme le pattern Template Method, limitent la flexibilité au moment de l'exécution. Une fois la variante de l'algorithme sélectionnée – dans notre exemple, c'est l'instance de `HTMLReport` –, il est difficile de changer d'avis. Pour modifier le format de sortie, le pattern Template Method nous contraint à créer un nouvel objet, par exemple `PlainTextReport`. Quelle est donc l'alternative ?

Déléguer, déléguer et encore déléguer

L'alternative est de suivre le conseil du GoF mentionné au Chapitre 1 : préférer la délégation. Et si au lieu de créer une sous-classe pour chaque variante on extrayait le

fragment de code variable qui nous ennuie pour l'encapsuler dans sa propre classe ? On pourrait définir toute une famille de classes, une pour chaque variante. Voici notre code de formatage HTML de l'exemple précédent, transplanté dans sa propre classe :

```
class Formatter
  def output_report( title, text )
    raise 'Abstract method called'
  end
end

class HTMLFormatter < Formatter
  def output_report( title, text )
    puts('<html>')
    puts('<head>')
    puts(" <title>#{title}</title>")
    puts(' </head>')
    puts(' <body>')
    text.each do |line|
      puts(" <p>#{line}</p>" )
    end
    puts(' </body>')
    puts('</html>')
  end
end
```

Notre classe de formatage du texte simple :

```
class PlainTextFormatter < Formatter
  def output_report(title, text)
    puts("***** #{title} *****")
    text.each do |line|
      puts(line)
    end
  end
end
```

Maintenant que les détails du formatage sont supprimés de la classe `Report`, elle devient beaucoup plus simple :

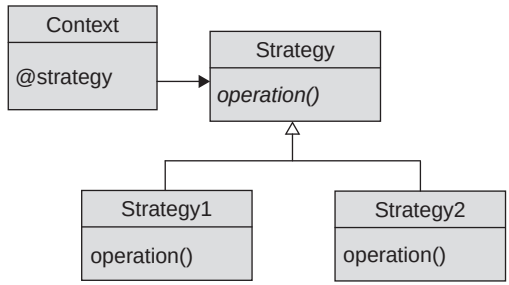
```
class Report
  attr_reader :title, :text
  attr_accessor :formatter
  def initialize(formatter)
    @title = 'Monthly Report'
    @text = [ 'Things are going', 'really, really, really well.']
    @formatter = formatter
  end

  def output_report
    @formatter.output_report( @title, @text )
  end
end
```

L'utilisation de la classe `Report` est légèrement plus compliquée. Il faut maintenant fournir à la classe `Report` un objet de formatage correspondant :

```
report = Report.new(HTMLFormatter.new)
report.output_report
```

Les membres du GoF ont baptisé la technique qui consiste à extraire l'algorithme dans un objet séparé du nom de pattern Strategy (voir Figure 4.1). L'idée sous-jacente du pattern Strategy est de définir une famille d'objets – de *stratégies* – ayant la même mission. Dans notre exemple, la mission est le formatage du rapport. Non seulement les stratégies effectuent le même travail, mais elles doivent présenter exactement la même interface. Dans notre exemple, les deux stratégies fournissent la méthode `output_report`.

Figure 4.1
Le pattern Strategy



Puisque les stratégies ont une interface identique, l'objet utilisateur – nommé par le GoF la classe de *contexte* – peut les traiter comme des pièces interchangeables. Peu importe quelle stratégie vous appelez, elles exposent la même interface pour le monde extérieur et elles exécutent la même fonction.

Mais il est important de choisir la bonne stratégie, car chacune d'elles fait le traitement différemment. Dans notre exemple, l'une des stratégies de formatage produit du HTML tandis que l'autre affiche du texte simple. Si l'on faisait des calculs d'impôts pour les résidents des différents pays européens, on pourrait utiliser le pattern Strategy en fonction des règles de calcul d'impôts sur le revenu dans les différents États : une stratégie pour calculer les impôts des résidents de la France et une autre pour l'Allemagne.

Le pattern Strategy possède de vrais avantages. Comme nous l'avons vu dans l'exemple du rapport, l'extraction des stratégies d'une classe permet d'atteindre une meilleure séparation des fonctionnalités. Avec le pattern Strategy nous libérons la classe `Report` de toute responsabilité et de toute connaissance du format.

Le pattern Strategy est fondé sur la composition et la délégation plutôt que l'héritage, il est donc facile d'alterner des stratégies au moment de l'exécution. Il suffit de remplacer l'objet stratégie :

```
report = Report.new(HTMLFormatter.new)
report.output_report
report.formatter = PlainTextFormatter.new
report.output_report
```

Le pattern Strategy a une caractéristique en commun avec le pattern Template Method. Les deux nous permettent de concentrer à un seul endroit la décision sur la variante de l'algorithme à utiliser. Avec le pattern Template Method on prend la décision lorsque l'on sélectionne une sous-classe concrète. Avec Strategy on choisit la classe stratégie au moment de l'exécution.

Partager les données entre l'objet contexte et l'objet stratégie

Un atout significatif du pattern Strategy est qu'un mur de données sépare élégamment les objets contexte et stratégie, puisque leur code réside dans des classes différentes. La mauvaise nouvelle est que nous devons inventer un moyen de faire transiter à travers ce mur les informations du contexte nécessaires à la stratégie. Concrètement, nous avons deux options. Premièrement, nous pouvons continuer avec l'approche adoptée jusqu'alors : passer tout ce dont l'objet stratégie a besoin en argument lorsque l'objet contexte appelle des méthodes de la stratégie. Rappelez-vous notre exemple : l'objet `Report` passait toute l'information requise par le gestionnaire de format dans les arguments de la méthode `output_report`. Le passage des données a l'avantage de garder le contexte et la stratégie parfaitement séparés. Les stratégies exposent une interface, le contexte l'utilise. L'inconvénient de cette approche, c'est que cela oblige à transférer beaucoup de données complexes entre le contexte et la stratégie sans aucune assurance qu'elles seront effectivement utilisées.

Deuxièmement, une stratégie peut récupérer les données du contexte si l'objet contexte lui-même est passé en argument par référence. L'objet stratégie peut ensuite appeler les méthodes du contexte pour accéder à toutes les données nécessaires. Voici ce que nous pouvons faire dans notre exemple :

```
class Report
  attr_reader :title, :text
  attr_accessor :formatter
  def initialize(formatter)
    @title = 'Monthly Report'
    @text = ['Things are going', 'really, really well.']
```

```
    @formatter = formatter
  end

  def output_report
    @formatter.output_report(self)
  end
end
```

L'objet `Report` est passé par référence à l'objet stratégie. Ensuite, la classe `formatter` appelle les méthodes `title` et `text` afin de récupérer les données nécessaires. Voici la classe `HTMLFormatter refactorisée` pour accompagner cette classe `Report` qui se passe elle-même en argument :

```
class Formatter
  def output_report(context)
    raise 'Abstract method called'
  end
end

class HTMLFormatter < Formatter
  def output_report(context)
    puts('<html>')
    puts(' <head>')
    puts(" <title>#{context.title}</title>")
    puts(' </head>')
    puts(' <body>')
    context.text.each do |line|
      puts(" <p>#{line}</p>")
    end
    puts(' </body>')
    puts('</html>')
  end
end
```

Alors que cette technique facilite la transmission de données entre le contexte et la stratégie, elle augmente également le couplage entre les deux parties. Cela amplifie le risque que les classes du contexte et les stratégies se mélangent.

Typage à la canard une fois de plus

Le programme de formatage que nous avons écrit reflète l'approche adoptée par le GoF concernant le pattern Strategy. Notre famille des stratégies de formatage est composée de la classe parent "abstraite" `Formatter` et des deux sous-classes `HTMLFormatter` et `PlainTextFormatter`. Toutefois, cette implémentation est "antiRuby". La classe `Formatter` ne sert à rien, elle n'existe que pour définir l'interface commune pour les sous-classes de formatage. Il n'y a pas d'anomalie au niveau du fonctionnement de cette approche, le programme fonctionne correctement. Néanmoins, ce type de code s'oppose à la philosophie rubyesque du typage à la canard. Les canards pourraient

affirmer (en couinant ?) que `HtmlFormatter` et `PlainTextFormatter` partagent déjà une interface commune car tous deux implémentent la méthode `output_report`. Il n'y a donc aucune raison de faire du sur-place et de créer une classe parent inutile.

Nous pouvons nous débarrasser de la classe `Formatter` en appuyant sur la touche "supprimer". Voici le code que l'on obtient :

```
class Report
  attr_reader :title, :text
  attr_accessor :formatter
  def initialize(formatter)
    @title = 'Monthly Report'
    @text = ['Things are going', 'really, really well.']
    @formatter = formatter
  end

  def output_report()
    @formatter.output_report(self)
  end
end

class HTMLFormatter
  def output_report(context)
    puts('<html>')
    puts(' <head>')
    # Imprimer le reste du rapport ...
    puts(" <title>#{context.title}</title>")
    puts(' </head>')
    puts(' <body>')
    context.text.each do |line|
      puts(" <p>#{line}</p>")
    end
    puts(' </body>')
    puts('</html>')
  end
end

class PlainTextFormatter
  def output_report(context)
    puts("***** #{context.title} *****")
    context.text.each do |line|
      puts(line)
    end
  end
end
```

Si nous comparons ce code à celui de la version précédente, nous verrons que la classe parent `Formatter` est éliminée sans provoquer de changements. Grâce au typage dynamique nous obtenons toujours les rapports annonçant que tout va bien. Malgré un fonctionnement correct des deux versions présentées, le monde Ruby voterait sans ambiguïté pour la suppression de la classe `Formatter`.

Procs et blocs

Il s'avère qu'effacer la classe parent n'est pas la seule façon de refondre le pattern Strategy à la Ruby. Mais avant de procéder à l'étape suivante nous devons explorer un des aspects les plus intéressants de Ruby : les blocs de code et l'objet `Proc`.

En tant qu'utilisateur des langages de programmation orientés objet, nous réfléchissons beaucoup sur des objets et comment ils interagissent. Mais notre vision des objets est quelque peu asymétrique. Nous pouvons facilement séparer les données d'un objet – on peut aisément extraire `@text` de l'objet `report` et l'utiliser indépendamment de l'objet `report`. Pourtant, nous considérons que notre code est intimement lié et inséparable de l'objet auquel il est attaché. Évidemment, ce n'est pas la seule façon de fonctionner. Et si nous pouvions sortir des fragments de code et les passer comme s'ils étaient des objets ? Ruby met à notre disposition les moyens d'y parvenir.

Un `Proc` est un objet Ruby qui contient un fragment de code. La méthode `lambda` est la manière la plus courante de créer un `Proc` :

```
hello = lambda do
  puts('Hello')
  puts('I am inside a proc')
end
```

En Ruby, le fragment de code se trouvant entre `do` et `end` se nomme *un bloc*[1]. La méthode `lambda` renvoie un objet `Proc`, qui est un conteneur pour le code entouré de `do` et `end`. Notre variable `hello` est une référence vers l'objet `Proc`. Nous pouvons exécuter le code incorporé dans l'objet `Proc` en appelant (quoi d'autre ?) la méthode `call`. Si l'on appelle la méthode `call` de notre objet `Proc`

```
hello.call
```

on obtient

```
Hello
I am inside a proc
```

De façon extrêmement utile, un objet `Proc` récupère le contexte de son environnement d'exécution au moment de sa création. Toutes les variables visibles au moment de la création d'un `Proc` lui demeurent accessibles à l'exécution. Par exemple, il n'y a qu'un seul nom de variable dans le code suivant :

```
name = 'John'
```

1. Les autres noms de ce concept sont *une fermeture lexicale* et *lambda*, d'où le nom de notre méthode qui fabrique un `Proc`.

```
proc = Proc.new do
  name = ' Mary'
end
proc.call
puts(name)
```

À l'exécution, la variable `name` recevra la valeur "John" dans la première instruction, ensuite, le `Proc` lui affectera la valeur "Mary". Enfin, Ruby affichera "Mary".

Si vous trouvez que `do` et `end` représentent trop de code à taper, Ruby propose une syntaxe plus concise avec des accolades. Voici une façon plus rapide de créer le `Proc` "hello" :

```
hello = lambda {
  puts('Hello, I am inside a proc')
}
```

Les programmeurs Ruby ont adopté une convention qui consiste à utiliser `do/end` pour des blocs sur plusieurs lignes et des accolades pour ceux qui tiennent sur une ligne[1]. Voici une version de notre exemple plus conforme à la convention :

```
hello = lambda {puts('Hello, I am inside a proc')}
```

Les objets `Proc` ont beaucoup de points communs avec les méthodes. Par exemple, non seulement des objets `Proc`, tout comme des méthodes, sont des fragments de code, mais les deux peuvent retourner une valeur. Un `Proc` retourne toujours la dernière valeur évaluée dans le bloc. Pour retourner une valeur à partir d'un objet `Proc` il faut donc s'assurer qu'elle soit évaluée par la dernière instruction de cet objet. La méthode `call` retransmet la valeur renvoyée par un `Proc`. Par conséquent, le code suivant

```
return_24 = lambda {24}
puts(return_24.call)
```

affiche

```
24
```

Vous pouvez également définir des paramètres à passer à votre objet `Proc`, bien que la syntaxe soit un peu étrange. Au lieu d'entourer la liste de paramètres des parenthèses habituelles "()", on ouvre et ferme la liste par une barre verticale "|" :

```
multiply = lambda { |x, y| x * y}
```

1. En réalité, il existe une différence entre les opérateurs `do/end` et les accolades. Dans les instructions Ruby les accolades ont une priorité supérieure par rapport à `do/end`, elles sont évaluées avant le reste. Cette différence ne devient visible que lorsque l'on commence à omettre des parenthèses optionnelles.

Ce code définit un objet `Proc` qui accepte deux paramètres, les multiplie et retourne le résultat. Pour appeler un `Proc` avec des paramètres, on les passe simplement à la méthode `call` :

```
n = multiply.call(20, 3)
puts(n)
n = multiply.call(10, 50)
puts(n)
```

L'exécution de ce code affiche le résultat suivant :

```
60
500
```

La possibilité de placer des blocs de code à différents endroits de vos programmes est tellement utile que Ruby définit une syntaxe raccourcie spéciale. Si l'on veut passer un bloc dans une méthode, il suffit de l'ajouter à la fin de l'appel de cette méthode. La méthode pourra ensuite exécuter le bloc à l'aide du mot clé `yield`. Voici un exemple d'une méthode qui affiche un message, exécute un bloc, puis affiche un deuxième message :

```
def run_it
  puts("Before the yield")
  yield
  puts("After the yield")
end
```

Le code ci-après appelle `run_it`. Remarquez que nous ajoutons simplement le bloc de code à la fin de l'appel de méthode :

```
run_it do
  puts('Hello')
  puts('Coming to you from inside the block')
end
```

Lorsque l'on colle un bloc de code à l'appel de méthode, le bloc (qui est en réalité un objet `Proc`) est passé en tant qu'une sorte de paramètre invisible. Le mot clé `yield` exécute ce paramètre. Par exemple, l'exécution du code ci-dessus produit le résultat suivant :

```
Before the yield
Hello
Coming to you from inside the block
After the yield
```

Si le bloc passé à la méthode accepte des paramètres, il faut les fournir à l'appel `yield`.

Par exemple, le code suivant

```
def run it with parameter
  puts('Before the yield')
  yield(24)
  puts('After the yield')
end

run_it_with_parameter do |x|
  puts('Hello from inside the proc')
  puts("The value of x is #{x}")
end
```

affiche

```
Before the yield
Hello from inside the proc
The value of x is 24
After the yield
```

Parfois, on a besoin de rendre le paramètre de type bloc explicite, c'est-à-dire capturer le bloc passé à la méthode sous forme d'un objet `Proc` dans un vrai paramètre. Pour y parvenir, nous pouvons ajouter un paramètre spécial à la fin de notre liste d'arguments. Ce paramètre doit être précédé par une esperluette. Sa valeur deviendra l'objet `Proc` créé à partir du bloc de code que l'on a passé à l'appel de méthode. Voici une version équivalente de la méthode `run_it_with_parameter` :

```
def run it with parameter(&block)
  puts('Before the call')
  block.call(24)
  puts('After the call')
end
```

L'esperluette fonctionne également lors de l'utilisation inverse. Si l'on a un objet `Proc` dans une variable et qu'il doive être passé à une méthode qui s'attend à recevoir un bloc de code, nous pouvons ajouter une esperluette devant l'objet `Proc` pour le convertir en un bloc :

```
my_proc = lambda {|x| puts("The value of x is #{x}")}
run_it_with_parameter(&my_proc)
```

Stratégies rapides et pas très propres

Quel est le rapport entre cette histoire de blocs et de `Procs` et le pattern Strategy ? Tout simplement, une stratégie est un fragment de code exécutable qui sait accomplir une tâche – par exemple formater du texte – et qui est encapsulé dans un objet. Cette définition doit vous rappeler quelque chose car elle correspond aussi à celle d'un `Proc`, un fragment de code incorporé dans un objet.

Il est trivial de refondre notre programme de formatage pour qu'il se fonde sur la stratégie `Proc`. Ajouter une esperluette dans la méthode `initialize`, afin qu'elle accepte un bloc de code en argument, et renommer la méthode `output_report` en `call` sont les seules modifications à apporter :

```
class Report
  attr_reader :title, :text
  attr_accessor :formatter
  def initialize(&formatter)
    @title = 'Monthly Report'
    @text = [ 'Things are going', 'really, really well.' ]
    @formatter = formatter
  end

  def output_report
    @formatter.call( self )
  end
end
```

Il y a un peu plus de travail pour construire les objets de formatage. Nous devons désormais créer des objets `Proc` au lieu des instances de nos classes de formatage spéciales :

```
HTML_FORMATTER = lambda do |context|
  puts ( ' <html> ' )
  puts(' <head>')
  puts(" <title>#{context.title}</title>")
  puts(' </head>')
  puts(' <body>')
  context.text.each do |line|
    puts(" <p>#{line}</p>" )
  end
  puts(' </body>')
  puts
```

Nous voilà prêts à créer des rapports avec nos nouveaux objets de formatage fondés sur des `Procs`. Puisque nous avons un objet `Proc` et que le constructeur de la classe `Report` s'attend à recevoir un bloc de code, il faut ajouter une esperluette devant notre objet `Proc` lorsque l'on instancie un nouveau `Report` :

```
report = Report.new &HTML_FORMATTER
report.output_report
```

Pourquoi prendre la peine d'adopter la stratégie fondée sur des `Procs` ? Premièrement, nous ne sommes plus obligés de créer des classes spéciales pour notre stratégie, il suffit d'encapsuler le code dans un objet `Proc`. Plus important encore, on peut maintenant créer une stratégie comme par magie en passant un bloc de code directement dans une méthode. Par exemple, voici notre code de traitement du texte simple remanié pour devenir un bloc de code généré à la volée :

```
report = Report.new do |context|
  puts("***** #{context.title} *****")
  context.text.each do |line|
    puts(line)
  end
end
```

Les blocs de code peuvent paraître bizarres si vous n'y êtes pas habitué. Il faut noter que ces blocs de code nous ont aidés à simplifier grandement le pattern Strategy : nous avons condensé une classe contexte, une classe parent de stratégie et quelques stratégies concrètes avec leurs instances associées en une classe contexte et quelques blocs de code.

Est-ce que cela signifie qu'il faut simplement oublier les stratégies fondées sur des classes ? Pas vraiment. Les stratégies en forme de blocs fonctionnent uniquement lorsque l'interface de la stratégie est très simple et qu'elle ne dispose que d'une seule méthode. Après tout, la seule méthode que l'on peut appeler sur un objet `Proc` est `call`. Si votre stratégie est plus sophistiquée, il ne faut pas hésiter à définir quelques classes. Mais si vos demandes peuvent être satisfaites par une stratégie simple, un bloc de code est probablement la bonne solution.

User et abuser du pattern Strategy

La façon la plus simple de se tromper avec le pattern Strategy est de faire des erreurs dans l'interface entre le contexte et la stratégie. Rappelez-vous que vous essayez de récupérer une tâche cohérente et plus ou moins autonome de l'objet contexte et de la déléguer à la stratégie. Il faut bien veiller aux détails de l'interface entre le contexte et la stratégie ainsi qu'à leur couplage. N'oubliez pas que le pattern Strategy ne vous sera pas utile si vous couplez le contexte avec la première stratégie jusqu'au point de ne pas pouvoir insérer une deuxième ou troisième stratégie dans le système.

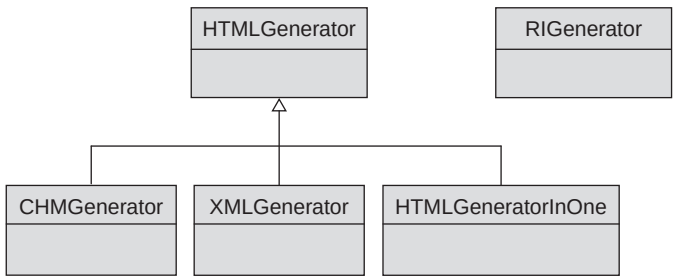
Le pattern Strategy dans le monde réel

L'utilitaire rdoc, livré en standard avec votre distribution Ruby, contient quelques instances du pattern Strategy classique, fondé sur des classes. Le but de rdoc est d'extraire la documentation des programmes. À part Ruby, rdoc est capable de traiter la documentation des programmes en C et (Dieu nous garde...) en FORTRAN. L'utilitaire rdoc utilise le pattern Strategy pour traiter chacun des langages de programmation : il contient un analyseur C, un analyseur Ruby et un analyseur FORTRAN. Chacun d'eux est une stratégie permettant de gérer leur format d'entrée respectif.

L'utilitaire rdoc vous offre également un choix de formats de sortie. On peut générer la documentation en plusieurs variantes de HTML, XML ou en format utilisé nativement

par la commande Ruby `ri`. Vous avez probablement deviné que chacun des formats de sortie est géré par sa propre stratégie. La relation entre les différentes stratégies de rdoc démontre l'attitude typique de Ruby envers l'héritage. Cette relation est illustrée à la Figure 4.2.

Figure 4.2
Les générateurs de rdoc



Il existe quatre stratégies de sortie apparentées – rdoc les appelle des générateurs – et une stratégie indépendante (voir Figure 4.2). Les quatre stratégies apparentées génèrent des fichiers similaires : du texte `<stuff>` entouré des balises entre crochets `</stuff>`[1]. La dernière stratégie génère les fichiers de sortie pour la commande Ruby `ri`. Ce format ne ressemble ni à du XML ni à du HTML. Comme vous pouvez le voir dans le diagramme UML de la Figure 4.2, la relation entre les classes reflète les choix de l'implémentation : les classes qui gèrent le HTML, le CHM et le XML ont de par la nature de leurs formats beaucoup de code en commun et sont donc liées par l'héritage. La classe `RIGenerator` produit quant à elle un format de sortie totalement différent et n'a aucun lien avec la famille XML/HTML. Le code de rdoc ne crée donc pas de classe parent commune pour imposer la même interface à tous les générateurs. Chaque générateur implémente les bonnes méthodes et c'est tout.

Nous avons sous la main un bon exemple d'un objet `Proc` utilisé en tant que stratégie légère. C'est notre bon vieux `tableau`. Si l'on veut trier le contenu d'un tableau Ruby, on appelle simplement la méthode `sort` :

```
a = ['russell', 'mike', 'john', 'dan', 'rob']
a.sort
```

Par défaut, la méthode `sort` trie le tableau dans l'ordre "naturel" de ses éléments. Et si nous voulions définir un algorithme de tri différent ? Par exemple, si nous voulions trier

1. CHM est un type de HTML utilisé par les fichiers d'aide Microsoft. Notez que c'est le code rdoc et pas moi qui suggère que XML est une sorte de HTML.

en fonction de la longueur des chaînes de caractères ? Nous passons simplement une stratégie de comparaison en forme de bloc de code :

```
a.sort { |a,b| a.length <=> b.length }
```

La méthode `sort` appellera votre bloc lorsqu'elle devra comparer deux éléments d'un tableau. Le bloc doit retourner `1` si le premier élément est supérieur, `0` si les deux sont égaux et `-1` si c'est le second élément qui est supérieur. C'est exactement ce que fait l'opérateur <=> et ce n'est pas une coïncidence.

En conclusion

Le pattern Strategy est un modèle fondé sur la délégation. Son but est de proposer une solution au même problème que celui résolu par le pattern Template Method. Au lieu d'extraire des parties variables de votre algorithme et de les encapsuler dans des sous-classes, vous implémentez chaque version de l'algorithme dans un objet séparé. Ensuite, varier l'algorithme consiste à fournir les différents objets stratégie à l'objet contexte – par exemple une stratégie pour produire du HTML, une autre pour générer des fichiers PDF ; ou une stratégie pour calculer des impôts en France et une autre pour des impôts en Italie.

Nous avons plusieurs options pour récupérer les données nécessaires dans l'objet contexte et les transmettre à l'objet stratégie. On peut passer toutes les données en paramètre lorsque l'on appelle les méthodes de l'objet stratégie, ou simplement passer à la stratégie une référence vers l'objet contexte entier.

Les blocs de code Ruby, qui représentent essentiellement des fragments de code encapsulés dans un objet `Proc`, sont merveilleusement utiles pour créer rapidement des objets stratégie simples.

Comme nous allons le voir dans les chapitres suivants, le pattern Strategy ressemble, au moins au premier regard, à plusieurs autres patterns. Par exemple, dans le pattern Strategy nous avons un objet – le contexte – qui essaie d'accomplir une tâche. Pour y arriver nous devons lui fournir un deuxième objet – la stratégie – qui aide à faire le traitement nécessaire. Vu de manière superficielle, le pattern Observer semble fonctionner de façon similaire : un objet fait un travail, mais au cours de l'exécution il fait appel à un deuxième objet que nous devons fournir.

La différence entre les deux patterns est dans la nature de l'intention. Le but du pattern Strategy est de donner au contexte l'accès à un objet qui est au courant d'une variation de l'algorithme. Le but du pattern Observer est très différent : l'intention est... mais nous devrions peut-être laisser cette distinction pour un autre chapitre, le suivant.

5

Rester informé avec le pattern Observer

Un des défis les plus ardus de la conception réside dans le développement d'un système complètement coordonné, c'est-à-dire un système dans lequel chaque partie reste informée de l'état de l'ensemble du système. Pensez à un logiciel de type tableur : modifier la valeur d'une cellule n'affecte pas seulement cette valeur particulière mais elle est aussi répercutée dans les totaux de la colonne, modifie la hauteur d'une des barres de l'histogramme et active le bouton "enregistrer". Un exemple encore plus simple : un système de gestion de ressources humaines doit notifier le département comptabilité lorsque le salaire d'une personne est modifié.

Construire ce type de système est assez difficile. Ajoutez-y l'impératif de garder le système maintenable et la tâche devient vraiment compliquée. Comment coordonner les composants indépendants d'un grand système logiciel sans augmenter le couplage entre les classes au point de rendre l'ensemble totalement ingérable ? Comment développer un tableur qui affiche des changements de données correctement sans coder en dur le lien entre le code d'édition et le constructeur d'histogrammes ? Comment permettre à l'objet `Employee` de passer les nouvelles concernant les changements de son salaire sans mélanger son code avec le composant de comptabilité ?

Rester informé

Une façon d'approcher le problème consiste à se concentrer sur le fait que la cellule du tableau ainsi que l'objet `Employee` sont tous les deux à l'origine d'une notification. Fred est augmenté et son enregistrement `Employee` annonce au monde entier (ou au moins à tous ceux qui sont intéressés) : "Bonjour ! Il y a eu du changement ici !" Tout

objet qui s'intéresse à l'état des finances de Fred doit s'être enregistré auprès de l'objet `Employee` au préalable. Tout objet enregistré recevra en temps opportun les notifications concernant les fluctuations des gains de Fred.

Comment exprimer tout ceci dans le code ? Voici la version de base de l'objet `Employee`. Il ne contient pas de code pour réactualiser d'autres objets, pour l'instant il s'occupe uniquement de tenir à jour les informations d'un salarié :

```
class Employee
  attr_reader :name
  attr_accessor :title, :salary
  def initialize( name, title, salary )
    @name = name
    @title = title
    @salary = salary
  end
end
```

Puisque le champ `salary` est modifiable grâce à son accesseur `attr_accessor`, nos salariés peuvent être augmentés[1] :

```
fred = Employee.new("Fred Flintstone", "Crane Operator", 30000.0)
# Augmenter Fred
fred.salary = 35000.0
```

Ajoutons du code (assez naïf) pour tenir le département comptabilité informé des changements du salaire :

```
class Payroll
  def update( changed_employee )
    puts("Cut a new check for #{changed_employee.name}!")
    puts("His salary is now #{changed_employee.salary}!")
  end
end

class Employee
  attr_reader :name, :title
  attr_reader :salary
  def initialize( name, title, salary, payroll)
    @name = name
    @title = title
    @salary = salary
    @payroll = payroll
end
```

1. Le salaire de nos employés peut aussi être réduit, mais nous allons fermer les yeux sur cette possibilité désagréable.

```
  def salary=(new_salary)
    @salary = new_salary
    @payroll.update(self)
  end
end
```

Nous pouvons maintenant changer la paie de Fred :

```
payroll = Payroll.new
fred = Employee.new('Fred', 'Crane Operator', 30000, payroll)
fred.salary = 35000
```

La comptabilité le saura aussitôt :

```
Cut a new check for Fred!
His salary is now 35000!
```

Puisque le département comptabilité doit être informé des changements du salaire, on ne peut pas utiliser la méthode `attr_accessor` pour le champ `salary`. Nous devons écrire à la main la méthode `salary=`.

Une meilleure façon de rester informé

Il y a un problème avec ce programme : le lien pour informer la comptabilité des changements du salaire est codé en dur. Et si nous devions tenir d'autres objets – par exemple d'autres classes de la comptabilité – au courant des finances de Fred ? La façon dont le code est conçu actuellement nous obligerait à modifier la classe `Employee` à nouveau, ce qui serait regrettable, car dans cette classe rien n'a réellement changé. Ce sont les autres classes – celle de la comptabilité – qui provoquent des changements dans `Employee`. Force est de constater que notre classe `Employee` semble être très peu résistante aux changements.

Nous devrions peut-être prendre du recul et résoudre ce problème de notifications de manière plus générique. Comment peut-on séparer les parties variables – les objets qui doivent apprendre la nouvelle sur les changements du salaire – du fonctionnement interne de l'objet `Employee` ? Il semble que nous ayons besoin de bâtir un tableau des objets intéressés par les dernières nouvelles de l'objet `Employee`. On pourrait créer ce tableau dans la méthode `initialize` :

```
def initialize( name, title, salary )
  @name = name
  @title = title
  @salary = salary
  @observers = []
end
```

Nous avons aussi besoin du code qui informera tous les observateurs dès qu'un changement a lieu :

```
def salary=(new_salary)
  @salary = new_salary
  notify_observers
end

def notify_observers
  @observers.each do |observer|
    observer.update(self)
  end
end
```

La partie variable la plus importante de `notify_observers` est `observer.update (self)`. Ce fragment de code appelle la méthode `update` sur chaque observateur[1] en lui annonçant que quelque chose – le salaire dans ce cas précis – a changé dans l'objet `Employee`.

Il ne nous reste qu'à écrire les méthodes pour ajouter et supprimer des observateurs de l'objet `Employee` :

```
def add_observer(observer)
  @observers << observer
end

def delete_observer(observer)
  @observers.delete(observer)
end
```

Désormais, tout objet qui s'intéresse aux changements du salaire de Fred peut s'enregistrer en tant qu'observateur de l'objet `Employee` de Fred :

```
fred = Employee.new('Fred', 'Crane Operator', 30000.0)
payroll = Payroll.new
fred.add_observer( payroll )
fred.salary = 35000.0
```

Ce mécanisme nous permet de supprimer le couplage implicite entre la classe `Employee` et l'objet `Payroll`. L'objet `Employee` ne se préoccupe plus du nombre d'objets intéressés par ses changements de salaire, il passe l'information à tout objet qui s'est déclaré concerné. L'objet `Employee` fonctionnerait tout aussi bien avec un seul, plusieurs ou aucun observateur.

---

1. Rappelez-vous que `array.each` est la terminologie Ruby pour exprimer une boucle qui parcourt tous les éléments d'un tableau. Nous en apprendrons plus sur `array.each` au Chapitre 7.

```
class TaxMan
  def update( changed_employee )
    puts("Send #{changed_employee.name} a new tax bill!")
  end
end
tax_man = TaxMan.new
fred.add_observer(tax_man)
```

Supposons que le salaire de Fred soit modifié une fois de plus :

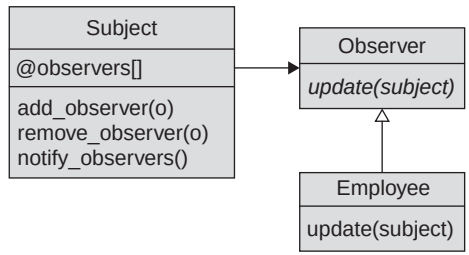
```
fred.salary = 90000.0
```

Maintenant, le département comptabilité ainsi que l'inspection des impôts seront tous les deux au courant :

```
Cut a new check for Fred!
His salary is now 90000.0!
Send Fred a new tax bill!
```

Les membres du GoF ont appelé cette idée de construire une interface propre entre une source d'information et ses consommateurs le pattern Observer (voir Figure 5.1). Ils ont nommé la classe génératrice de l'information la classe *sujet*. Dans notre exemple, `Employee` est le sujet. Les *observateurs* sont des objets qui souhaitent recevoir l'information. Dans notre exemple, il y en a deux : `Payroll` et `TaxMan`. Lorsqu'un objet doit rester informé de l'état du sujet, il s'enregistre en tant qu'observateur de ce sujet.

Figure 5.1
Le pattern Observer



Il m'a toujours semblé que le pattern Observer était mal nommé. L'objet observateur s'attribue tout le mérite alors que le sujet effectue la plupart du travail. La responsabilité de conserver le suivi des observateurs échoit au sujet. Le sujet est également chargé de tenir les observateurs au courant des changements qui se produisent. Dit autrement, il est bien plus difficile de publier et de distribuer un journal que de le lire.

Factoriser le code de gestion du sujet

Généralement, l'implémentation du pattern Observer en Ruby n'est pas plus compliquée que notre exemple avec `Employee` : il suffit d'avoir un tableau pour stocker les observateurs, une paire de méthodes pour gérer le tableau et la méthode pour notifier tout le monde lors des changements. Mais il y a sûrement un moyen de mieux faire et de ne pas répéter ce code à chaque fois que l'on souhaite rendre un objet "observable". On pourrait utiliser l'héritage. Le résultat de la factorisation du code pour gérer le sujet est une petite classe parent fonctionnelle :

```
class Subject
  def initialize
    @observers=[]
  end

  def add_observer(observer)
    @observers << observer
  end

  def delete_observer(observer)
    @observers.delete(observer)
  end

  def notify_observers
    @observers.each do |observer|
      observer.update(self)
    end
  end
end
```

Maintenant, `Employee` peut devenir une sous-classe de `Subject` :

```
class Employee < Subject
  attr_reader :name, :address
  attr_reader :salary
  def initialize( name, title, salary)
    super()
    @name = name
    @title = title
    @salary = salary
  end

  def salary=(new_salary)
    @salary = new_salary
    notify_observers
  end
end
```

C'est une solution assez raisonnable. D'ailleurs, Java a choisi d'aller précisément dans cette voie avec sa classe `java.util.Observable`. Mais, comme nous l'avons vu au Chapitre 1, l'héritage peut être une source de tracas. Le problème, avec l'usage de

`Subject` en tant que classe parent, c'est l'impossibilité d'avoir une classe parent différente. Ruby permet à chaque classe d'avoir une classe mère unique. Lorsque la classe `Employee` choisit d'étendre `Subject`, il ne lui reste aucune autre option. Si votre modèle d'objets requiert qu'`Employee` soit une sous-classe de `DatabaseObject` ou `OrganizationalUnit`, pas de chance, elle ne pourra plus devenir une sous-classe de `Subject`.

Parfois, nous avons besoin de partager du code entre des classes totalement indépendantes et cela peut poser un problème. Notre classe `Employee` voudrait être `Subject`, mais il est possible que des cellules du tableur souhaitent l'être aussi. Comment partager l'implémentation de `Subject` sans gaspiller le droit d'avoir une classe parent ?

La solution à ce dilemme est d'utiliser un module. Souvenez-vous qu'un module est un conteneur de méthodes et de constantes que l'on peut partager entre des classes sans recourir à l'utilisation de la seule et unique classe mère. Après sa refonte en module notre classe `Subject` n'est guère différente :

```
module Subject
  def initialize
    @observers=[]
  end

  def add_observer(observer)
    @observers << observer
  end

  def delete_observer(observer)
    @observers.delete(observer)
  end

  def notify_observers
    @observers.each do |observer|
      observer.update(self)
    end
  end
end
```

Notre nouveau module `Subject` est très simple d'utilisation. On inclut le module et on appelle `notify_observers` lors des changements :

```
class Employee
  include Subject
  attr_reader :name, :address
  attr_reader :salary
  def initialize( name, title, salary)
    super()
    @name = name
    @title = title
    @salary = salary
  end
```

```
  def salary=(new_salary)
    @salary = new_salary
    notify_observers
  end
end
```

L'inclusion du module `Subject` rend disponibles toutes ses méthodes à la classe `Employee` : elle est désormais prête à agir comme un sujet du pattern Observer. Remarquez que nous devons appeler `super()` dans la méthode `initialize` de la classe `Employee`, ce qui a pour effet d'appeler `initialize` du module `Subject`[1].

Développer notre propre module `Subject` était amusant et constituait un bon entraînement à l'écriture des modules mixin. Mais est-ce que son utilisation est vraiment justifiée ? Probablement pas. La bibliothèque standard Ruby comprend en effet un formidable module `Observable` qui fournit tous les outils nécessaires pour transformer vos objets en sujets du pattern Observer. Son usage ne diffère pas beaucoup de notre module `Subject` :

```
require 'observer'
class Employee
  include Observable
  attr_reader :name, :address
  attr_reader :salary
  def initialize( name, title, salary)
    @name = name
    @title = title
    @salary = salary
  end

  def salary=(new_salary)
    @salary = new_salary
    changed
    notify_observers(self)
  end
end
```

Le module standard `Observable` propose une fonctionnalité que nous avons oubliée dans notre version artisanale. Ce module vous oblige à appeler la méthode `changed` avant d'appeler `notify_observers` afin de réduire le nombre de notifications redondantes. La méthode `changed` affecte une variable booléenne pour indiquer qu'un changement a réellement eu lieu ; la méthode `notify_observers` n'envoie pas de

1. L'appel vers `super()` est un des rares cas en Ruby où les parenthèses autour d'une liste d'arguments vide sont obligatoires. L'appel effectué avec des parenthèses, comme nous le faisons dans l'exemple, provoque un appel sans arguments vers la méthode dans la classe parent. Si l'on omet les parenthèses, la classe parent sera appelée avec la liste initiale des arguments, dans ce cas `name`, `title`, `salary` et `payroll_manager`.

notifications tant que la valeur de la variable n'est pas à `true`. Chaque appel à `notify_observers` réinitialise cette variable avec la valeur `false`.

Des blocs de code comme observateurs

Les blocs de code Ruby vont une fois de plus permettre une variation intéressante sur le thème du pattern Observer. Le code est simplifié de manière significative lorsqu'on utilise un bloc comme écouteur. Étant donné que la classe `Observable` fournie avec la bibliothèque Ruby ne supporte pas les blocs de code, une version légèrement modifiée de notre module `Subject` pourrait finalement nous être utile :

```
module Subject
  def initialize
    @observers=[]
  end

  def add_observer(&observer)
    @observers << observer
  end

  def delete_observer(observer)
    @observers.delete(observer)
  end

  def notify_observers
    @observers.each do |observer|
      observer.call(self)
    end
  end
end

class Employee
  include Subject
  attr_accessor :name, :title, :salary
  def initialize( name, title, salary )
    super()
    @name = name
    @title = title
    @salary = salary
  end

  def salary=(new_salary)
    @salary = new_salary
    notify_observers
  end
end
```

L'avantage des blocs réside dans la simplicité du code car il n'est plus nécessaire de mettre en place une classe séparée pour nos observateurs. Pour en ajouter un, nous appelons simplement `add_observer` en lui passant un bloc de code :

```
fred = Employee.new('Fred', 'Crane Operator', 30000)
fred.add_observer do |changed_employee|
  puts("Cut a new check for #{changed_employee.name}!")
  puts("His salary is now #{changed_employee.salary}!")
end
```

Cet exemple passe un observateur sous la forme d'un bloc de deux lignes à l'objet `Employee`. Avant d'arriver dans l'objet `Employee` les deux lignes sont converties en un objet `Proc`, qui est prêt à jouer le rôle d'observateur. Lorsque le salaire de Fred change, l'objet `Employee` appelle la méthode `call` de l'objet `Proc`, ce qui exécute les deux commandes `puts`.

Variantes du pattern Observer

Les décisions clés à prendre lorsque l'on implémente le pattern Observer concernent l'interface entre le sujet et l'observateur. Pour rester simple, on peut adopter l'approche de notre exemple ci-dessus : définir dans l'observateur une seule méthode qui prend un argument unique, le sujet. Les membres du GoF ont nommé cette stratégie la méthode *pull*, car les observateurs doivent demander au sujet tous les détails sur ses changements. Une autre possibilité – logiquement nommée la méthode *push* – consiste à envoyer les détails des changements du sujet vers les observateurs :

```
observer.update(self, :salary_changed, old_salary, new_salary)
```

Nous pouvons même définir des méthodes de mise à jour différentes pour différents événements. Par exemple, nous pourrions disposer d'une méthode pour déclencher les notifications des changements du salaire

```
observer.update_salary(self, old_salary, new_salary)
```

et d'une autre pour les changements d'intitulé

```
observer.update_title(self, old_title, new_title)
```

L'avantage de ce surcroît de détails réside dans la diminution de la charge de travail nécessaire pour garder les observateurs informés des changements. Le défaut du modèle *push* tient au fait que tous les observateurs ne sont pas forcément concernés par tous les détails et, dans ce cas, l'effort déployé pour passer l'information est inutile.

User et abuser du pattern Observer

Le problème majeur du pattern Observer tourne autour des choix de la fréquence et du moment choisis pour envoyer les notifications. Parfois, le volume même des notifications peut devenir une difficulté. Par exemple, un observateur pourrait s'enregistrer

avec un objet sans se rendre compte que cet objet lance des milliers de notifications par seconde. La classe sujet peut réduire ce risque en évitant de diffuser des mises à jour redondantes. Une mise à jour d'un objet ne signifie pas forcément qu'une vraie modification s'est produite. Souvenez-vous de la méthode `salary=` de l'objet `Employee`. On ne devrait probablement pas notifier les observateurs si aucun changement n'a eu lieu :

```
def salary=(new_salary)
  old_salary = @salary
  @salary = new_salary
  if old_salary != new_salary
    changed
    notify_observers(self)
  end
end
```

La cohérence du sujet lorsqu'il informe ses observateurs est un autre problème potentiel. Imaginez que nous complétions notre classe d'exemple afin qu'elle notifie ses observateurs des changements de l'intitulé du poste d'un employé ainsi que de son salaire :

```
def title=(new_title)
  old_title = @title
  @title = new_title
  if old_title != new_title
    changed = true
    notify_observers(self)
  end
end
```

Maintenant, imaginez que Fred reçoive une promotion importante, donc une augmentation. On pourrait le coder de façon suivante :

```
fred = Employee.new("Fred", "Crane Operator", 30000)
fred.salary = 1000000
# Attention ! Etat incohérent !
fred.title = 'Vice President of Sales'
```

Le souci, c'est que pendant un court instant Fred serait le grutier le mieux payé du monde, car son augmentation arriverait avant la modification de son intitulé de poste. Cela serait négligeable si tous nos observateurs n'étaient pas à l'écoute et n'étaient pas affectés par cet état incohérent. On peut remédier à ce problème en retardant la notification jusqu'au moment où l'ensemble des changements rentre en cohérence :

```
# Ne pas informer les observateurs
fred.salary = 1000000
fred.title = 'Vice President of Sales'
# Notifier les observateurs maintenant !
fred.changes_complete
```

Un dernier point : faites attention aux observateurs qui se comportent mal. Nous avons utilisé l'analogie d'un sujet qui transmet des nouvelles à un observateur, mais en réalité ce n'est qu'un appel de méthode sur un autre objet. Que faire si l'observateur lance une exception en réponse à la notification de l'augmentation de Fred ? Simplement écrire dans le log d'erreurs et continuer ou bien prendre des mesures plus énergiques ? Il n'existe pas de solution standard, tout dépend de votre application et de la confiance que vous accordez à vos observateurs.

Le pattern Observer dans le monde réel

Le pattern Observer n'est pas difficile à trouver dans le code Ruby. Par exemple, il est utilisé dans `ActiveRecord`. Des clients d'`ActiveRecord` qui souhaitent rester au courant des créations, lectures, mises à jour et suppressions d'objets dans la base de données peuvent définir des observateurs de la façon suivante[1] :

```
class EmployeeObserver < ActiveRecord::Observer
  def after_create(employee)
    # Un nouvel employé est enregistré
  end

  def after_update(employee)
    # L'enregistrement de l'employé est mis à jour
  end

  def after_destroy(employee)
    # L'enregistrement de l'employé est supprimé
  end
end
```

Dans l'élégant exemple du pattern *Convention plutôt que Configuration* (voir Chapitre 18), vous verrez qu'`ActiveRecord` ne vous oblige pas à enregistrer un observateur. Il déduit du nom de la classe `EmployeeObserver` que son rôle consiste à observer les objets `Employee`.

On trouve un autre exemple d'observateur fondé sur des blocs de code dans REXML, une bibliothèque de traitement XML livrée avec la bibliothèque standard de Ruby. La classe REXML SAX2Parser est un analyseur de flux XML : il lit un fichier XML et vous laisse libre d'ajouter des observateurs permettant d'être notifié lorsqu'un élément XML particulier est traité.

---

1. La syntaxe `ActiveRecord::Observer` peut vous paraître étrange car nous n'en avons pas encore parlé. Cette syntaxe indique que la classe `Observer` est définie à l'intérieur d'un module et il faut utiliser `::` pour y accéder.

Malgré le fait que SAX2Parser supporte un style plus formel d'observateurs définis dans des classes séparées, vous pouvez également lui passer des blocs de code qui serviront d'observateurs :

```
require 'rexml/parsers/sax2parser'
require 'rexml/sax2listener'
#
# Créer un parser XML pour nos données
#
xml = File.read('data.xml')
parser = REXML::Parsers::SAX2Parser.new( xml )
#
# Ajouter des observateurs pour être au courant des débuts et fins
# des éléments
#
parser.listen( :start_element ) do |uri, local, qname, attrs|
  puts("start element: #{local}")
end
parser.listen( :end_element ) do |uri, local, qname|
  puts("end element #{local}")
end
#
# Parser le XML
#
parser.parse
```

Passez un fichier XML au programme ci-dessus et vous pourrez regarder passer les notifications des éléments du fichier.

En conclusion

Le pattern Observer vous permet de développer des composants qui restent au courant des activités des autres composants en évitant de coupler l'ensemble trop fortement pour se retrouver dans un code spaghetti. L'interface claire entre la source de l'information (le sujet) et les consommateurs de cette information (les observateurs) permet de transmettre les messages sans embrouiller le code.

Le travail principal pour implémenter le pattern Observer est effectué dans la classe O*bservable*. Ruby nous permet de factoriser ce mécanisme soit dans une classe parent soit dans un module, ce dernier étant la solution la plus courante. L'interface entre le sujet et l'observateur peut être aussi complexe que nécessaire mais, si vous développez un observateur simple, les blocs de code font très bien l'affaire.

Comme mentionné au Chapitre 4, le pattern Observer et le pattern Strategy se ressemblent. Les deux présentent un objet principal (le sujet dans le pattern Observer et le contexte dans le pattern Stratégie) qui émet des appels vers un autre objet (un

observateur ou une stratégie). La principale différence réside dans l'intention. Dans le cas de l'Observer, nous informons l'autre objet des événements qui se produisent dans l'objet sujet. Dans le cas du pattern Strategy, nous déléguons à notre objet stratégie une partie du traitement.

Aussi, le pattern Observer remplit plus ou moins la même fonction que les méthodes d'accrochage (hooks) du pattern Template Method. Les deux permettent de tenir un objet informé des événements en cours. La différence, c'est bien sûr que le pattern Template Method n'informe que des objets qui lui sont liés par héritage. Si la classe n'est pas une de ses sous-classes, elle ne recevra aucune notification de la méthode d'accrochage du pattern Template Method.

6

Assembler le tout à partir des composants avec Composite

Lorsque j'ai eu 11 ou 12 ans, j'ai développé une théorie de l'univers. Je venais d'apprendre l'existence du système solaire et des atomes, et les ressemblances entre les deux semblaient être plus qu'une coïncidence. Notre système solaire a des planètes qui tournent autour du soleil, tandis que l'atome (au moins au niveau de l'école élémentaire) a des électrons qui tournent autour du noyau. Je me souviens que je me demandais si notre monde entier n'était pas juste un électron quelconque dans un univers plus grand. Et peut-être existait-il un adolescent fantastiquement petit qui vivait sa vie dans un monde qui faisait partie d'un atome au bout de mon crayon.

Malgré le fait que ma théorie se fondait sur de la physique erronée, l'idée de choses construites à partir de sous-ensembles similaires est un concept puissant qui peut être appliqué à l'écriture de vos programmes. Dans ce chapitre nous allons nous intéresser au pattern Composite, un motif de conception qui nous aide à construire des objets complexes à partir de sous-objets plus petits qui peuvent à leur tour être composés de sous-sous-objets encore plus petits. Nous verrons comment appliquer le pattern Composite à des situations aussi diverses que des organigrammes et la mise en page d'interfaces graphiques. Et, qui sait, dans le cas improbable où ma vieille théorie de l'univers serait exacte, nous pourrions utiliser le pattern Composite comme un modèle de l'univers.

Le tout et ses parties

Développer des logiciels orientés objet est un processus qui consiste à combiner des objets relativement simples, tels que des entiers et des chaînes de caractères, en objets

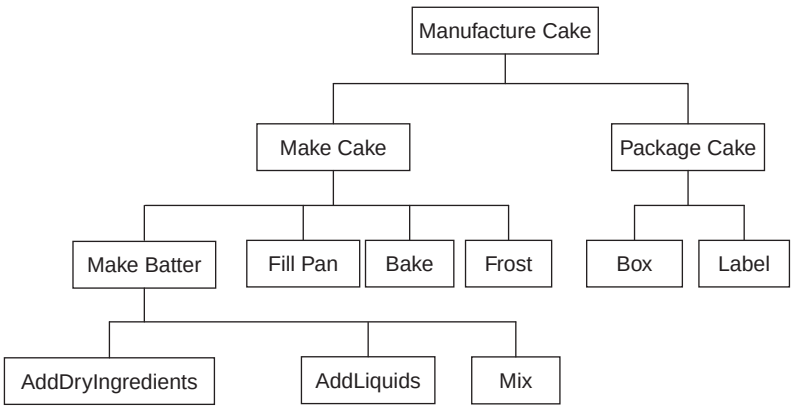
plus complexes, comme des dossiers personnels et des listes de chansons. Ces objets peuvent à leur tour être utilisés pour construire des objets encore plus intéressants. D'ordinaire, à la fin apparaissent quelques objets très sophistiqués qui ne ressemblent plus du tout aux composants qui ont servi à leur construction.

Mais ce n'est pas toujours le cas. Parfois, un objet complexe doit ressembler à ses composants et se comporter exactement de la même manière. Imaginez que vous travaillez dans la pâtisserie La Chocolatine SARL. Vous devez développer un système informatique qui suit le processus de fabrication des gâteaux au chocolat ChocOCroc. Un des prérequis indispensables est que votre système surveille le temps de préparation d'un gâteau. Évidemment, faire un gâteau est un processus assez complexe. D'abord, il faut préparer la pâte, la placer dans un moule et ensuite mettre le moule au four. Une fois le gâteau préparé il faut éventuellement le surgeler et le conditionner pour la vente. Préparer la pâte est aussi un processus assez compliqué qui consiste à mesurer la farine, ajouter des œufs et... bien sûr, lécher la cuillère.

Le processus de préparation d'un gâteau peut être considéré comme un arbre (voir Figure 6.1). La tâche principale "faire un gâteau" consiste en des sous-tâches telles que faire cuire le gâteau au four et l'emballer, qui peuvent, elles aussi, être divisées en sous-tâches.

Il est clair que nous ne voulons pas diviser le processus de fabrication d'un gâteau à l'infini ("Maintenant, ajoutez dans le réservoir le grain de farine numéro 1 463 474..."). Il faut se contenter d'identifier les tâches de bas niveau les plus fondamentales. Il est probablement judicieux de s'arrêter au niveau "ajouter des ingrédients secs" et "enfourner le gâteau" et de modéliser chacune de ces étapes dans une classe séparée.

Figure 6.1
L'arbre des tâches de préparation d'un gâteau



Il est évident que toutes les classes devront partager la même interface qui leur permettra de retourner l'information sur leur durée. Dans votre projet `La Chocolatine` vous décidez d'utiliser une classe parent commune `Task` et de créer quelques sous-classes, une pour chaque tâche basique : `AddDryIngredientsTask`, `AddLiquidsTask` et `MixTask`.

Assez parlé des tâches simples. Comment s'y prendre pour traiter des tâches plus complexes telles que "préparer la pâte" ou même "fabriquer un gâteau", qui consistent en des sous-tâches plus petites ? D'une part, ces sous-tâches sont parfaitement respectables. On pourrait vouloir savoir combien de temps est nécessaire pour préparer la pâte, faire l'emballage ou même fabriquer le gâteau entier exactement de la même manière qu'avec des tâches simples. Mais ces tâches de haut niveau n'ont pas tout à fait la même nature que des tâches simples : elles sont composées à partir d'autres tâches.

Vous aurez clairement besoin d'un conteneur pour traiter ces tâches complexes (ou devrions-nous dire *composites*). Il existe un autre point concernant les tâches de haut niveau dont il faut tenir compte : elles sont construites à partir d'un certain nombre de sous-tâches, mais de l'extérieur elles ont la même interface que n'importe quelle autre tâche `Task`.

Cette approche fonctionne à deux niveaux. Premièrement, le code utilisant l'objet `MakeBatterTask` n'est pas obligé de se préoccuper du fait que faire la pâte est plus compliqué que, par exemple, mesurer la farine. Que ce soit simple ou complexe, tout est un objet `Task`. C'est la même chose pour la classe `MakeBatter`, cette classe n'est pas concernée par les détails de ses sous-tâches ; qu'elles soient simples ou complexes, pour `MakeBatter` ce sont simplement des objets `Task`. Cela nous amène au deuxième point élégant de cette technique : `MakeBatter` gère simplement une liste d'objets `Task`, donc chacune des sous-tâches peut aussi consister en des sous-tâches. Enfin, nous pouvons construire un arbre de tâches et de sous-tâches aussi profond que nécessaire.

Il s'avère que la situation où l'on doit grouper des composants pour créer un nouveau supercomposant arrive assez fréquemment. Pensez à l'organisation dans des grandes entreprises. Toute entreprise est composée d'individus : des directeurs, des comptables et des ouvriers. Nous voyons rarement une grande entreprise comme une collection d'individus. Nous avons plutôt une vision d'un conglomérat de divisions qui sont séparées en départements eux-mêmes composés d'équipes dans lesquelles travaillent des individus.

Les départements et les divisions partagent beaucoup de caractéristiques avec les employés. Par exemple, chacun d'eux reçoit sa part de la masse salariale : Fred à la

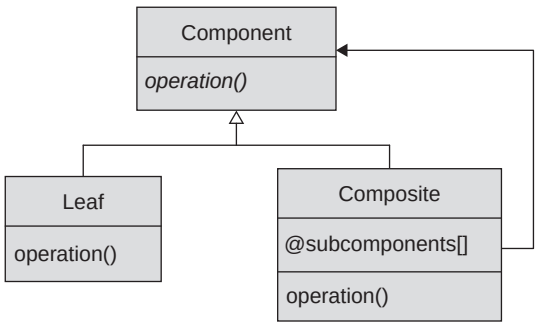
logistique peut être largement sous-payé, mais cela peut aussi être le cas du département des relations publiques. Les employés et les départements ont un responsable hiérarchique : Yves est le patron de Fred, tandis que Karine est la vice-présidente du département des relations publiques. Finalement, les départements peuvent quitter l'entreprise tout aussi bien que des employés : Fred peut trouver un travail mieux payé alors que le département des relations publiques peut être vendu à une autre entreprise. Du point de vue de la modélisation, les personnes dans une grande entreprise sont comparables à des tâches simples de la préparation d'un gâteau, tandis que les départements et les divisions sont des éléments de plus haut niveau, semblables à des tâches plus complexes de la préparation d'un gâteau.

Créer des composites

Pour désigner un système où "l'ensemble a le même comportement que ses parties", les membres du GoF ont créé le pattern Composite. Vous comprenez que vous avez besoin de ce pattern lorsque vous essayez de réaliser une hiérarchie ou un arbre d'objets et que vous ne voulez pas que le code client se demande constamment s'il a affaire à un seul objet où à toute une branche d'un grand arbre.

Trois éléments sont nécessaires pour construire le pattern Composite (voir Figure 6.2). Premièrement, il vous faut une interface ou une classe parent commune pour tous les objets. Dans la terminologie du GoF, cette classe parent ou interface s'appelle un *composant*. Posez-vous la question : "Quelles caractéristiques communes auront mes objets basiques et mes objet de haut niveau ?" Lorsque l'on prépare des gâteaux, les tâches simples comme mesurer la farine ainsi que les tâches complexes comme préparer la pâte prennent toutes deux un certain temps.

Figure 6.2
Le pattern Composite pattern



Deuxièmement, vous avez besoin des objets *feuilles*, des étapes simples et indivisibles du processus. Dans l'exemple avec le gâteau, les feuilles sont des tâches simples telles

que mesurer la farine ou ajouter des œufs. Dans l'exemple de l'organisation, les feuilles étaient des employés individuels. Les classes feuilles doivent, évidemment, implémenter l'interface `Component`.

Troisièmement, nous avons besoin d'au moins une classe de haut niveau, nommée par le GoF *composite*. L'objet composite est un composant, mais c'est aussi un objet de haut niveau qui consiste en des sous-composants. Dans l'exemple du gâteau, les composites sont des tâches complexes telles que préparer la pâte ou préparer le gâteau, ce sont des tâches qui consistent en des sous-tâches.

Pour des organisations, les objets composites sont des départements et des divisions.

Pour concrétiser cette discussion, jetons un œil sur le processus de préparation d'un gâteau exprimé en code. Commençons par la classe parent des composants :

```
class Task
  attr_reader :name
  def initialize(name)
    @name = name
  end

  def get_time_required
    0.0
  end
end
```

`Task` est une classe parent abstraite dans le sens où elle n'est pas complète. Elle conserve le nom de la tâche et fournit une méthode vide `get_time_required`.

Ajoutons deux classes feuilles :

```
class AddDryIngredientsTask < Task
  def initialize
    super('Add dry ingredients')
  end

  def get_time_required
    1.0#1 minute pour ajouter la farine et le sucre
  end
end

class MixTask < Task
  def initialize
    super('Mix that batter up!')
  end

  def get_time_required
    3.0# Remuer pendant 3 minutes
  end
end
```

Les classes `AddDryIngredientsTask` et `MixTask` sont des sous-classes très simples de `Task`. Elles exposent des implémentations de la méthode `get_time_required`. Évidemment, nous aurions pu continuer et définir toutes les tâches de base pour préparer un gâteau, mais utilisons notre imagination et allons droit au but avec la tâche composite :

```
class MakeBatterTask < Task
  def initialize
    super('Make batter')
    @sub_tasks = []
    add_sub_task( AddDryIngredientsTask.new )
    add_sub_task( AddLiquidsTask.new )
    add_sub_task( MixTask.new )
  end

  def add_sub_task(task)
    @sub_tasks << task
  end

  def remove_sub_task(task)
    @sub_tasks.delete(task)
  end

  def get_time_required
    time=0.0
    @sub_tasks.each {|task| time += task.get_time_required}
    time
  end
end
```

Alors que la classe `MakeBatterTask` apparaît de l'extérieur comme une tâche simple ordinaire – car elle implémente la méthode clé `get_time_required` –, elle se compose en fait de trois sous-tâches : `AddDryIngredientsTask` et `MixTask`, que nous avons déjà vues, et la tâche `AddLiquidsTask`, dont l'implémentation est laissée à votre imagination. La partie critique de `MakeBatterTask` est la façon dont cette tâche gère la méthode `get_time_required`. Pour être précis, `MakeBatterTask` additionne les durées de chaque sous-tâche.

Puisque dans notre exemple culinaire nous avons plusieurs tâches composites (emballage du gâteau et la tâche principale : préparation du gâteau), il serait judicieux de factoriser les détails de gestion des sous-tâches dans une autre classe parent :

```
class CompositeTask < Task
  def initialize(name)
    super(name)
    @sub_tasks = []
  end
```

```
    def add_sub_task(task)
      @sub_tasks << task
    end

    def remove_sub_task(task)
      @sub_tasks.delete(task)
    end

    def get_time_required
      time=0.0
      @sub_tasks.each { task| time += task.get_time_required}
      time
    end
  end
```

Notre classe `MakeBatterTask` est désormais réduite à ce qui suit :

```
class MakeBatterTask < CompositeTask
  def initialize
    super('Make batter')
    add_sub_task( AddDryIngredientsTask.new )
    add_sub_task( AddLiquidsTask.new )
    add_sub_task( MixTask.new )
  end
end
```

La profondeur illimitée de l'arbre est la caractéristique principale des objets composites. La classe `MakeBatterTask` n'a qu'un niveau de profondeur, mais ce ne sera jamais le cas en règle générale. Lorsque nous terminons notre projet culinaire, nous devrons créer la classe `MakeCake` :

```
class MakeCakeTask < CompositeTask
  def initialize
    super('Make cake')
    add_sub_task( MakeBatterTask.new )
    add_sub_task( FillPanTask.new )
    add_sub_task( BakeTask.new )
    add_sub_task( FrostTask.new )
    add_sub_task( LickSpoonTask.new )
  end
end
```

Chacune des sous-tâches de `MakeCakeTask` peut être un composite comme c'est le cas pour `MakeBatterTask`.

Raffiner le pattern Composite avec des opérateurs

On pourrait rendre le code de notre composite encore plus lisible si l'on remarquait le double rôle des objets composites. D'une part, un objet composite est un composant, d'autre part, c'est une collection de composants. Notre implémentation fait que la

classe `CompositeTask` ne ressemble pas beaucoup à des collections standard de Ruby, telles qu'un tableau ou un tableau associatif. Par exemple, ce serait agréable de pouvoir ajouter des tâches à `CompositeTask` à l'aide de l'opérateur `<<`, comme si c'était un tableau :

```
composite = CompositeTask.new('example')
composite << MixTask.new
```

Il s'avère que c'est très simple à réaliser si l'on renomme la méthode `add_sub_task` :

```
def <<(task)
  @sub_tasks << task
end
```

On pourrait aussi vouloir accéder aux sous-tâches en utilisant la syntaxe indicielle familière, comme dans le code suivant :

```
puts(composite[0].get_time_required)
composite[1] = AddDryIngredientsTask.new
```

Ruby traduira l'appel `object[i]` en un appel à la méthode répondant au nom étrange `[ ]`, qui accepte un seul paramètre, l'indice. Afin que notre classe `CompositeTask` supporte cette opération, il suffit d'y ajouter la méthode :

```
def [](index)
  @sub_tasks[index]
end
```

De la même manière, l'appel `object[i] = value` est traduit en un appel à la méthode avec le nom encore plus étrange `[]=`, qui accepte deux paramètres, l'indice et la nouvelle valeur :

```
def []=(index, new_value)
  @sub_tasks[index] = new_value
end
```

Un tableau comme composite ?

On pourrait également avoir le même conteneur si notre `CompositeTask` était une sous-classe d'`Array` plutôt que de `Task` :

```
class CompositeTask < Array
  attr_reader :name
  def initialize(name)
    @name = name
  end
```

```
    def get_time_required
      time=0.0
      each {|task| time += task.get_time_required}
      time
    end
  end
```

Étant donné le typage dynamique de Ruby, cette approche fonctionnera. Lorsque l'on transforme `CompositeTask` en une sous-classe d'`Array`, on obtient par l'héritage le conteneur et ses opérateurs associés `[]`, `[]=` et `<<`. Mais est-ce une bonne approche ?

Je vote contre. `CompositeTask` est non pas une espèce spécialisée d'un tableau mais une espèce spécialisée de `Task`. Si la classe `CompositeTask` doit être liée par héritage avec une autre classe, cette classe devrait être `Task` et non pas `Array`.

Une différence embarrassante

Toute implémentation du pattern Composite doit gérer un problème délicat. Nous avons commencé par dire que le but du pattern Composite est de rendre les objets feuilles plus ou moins indiscernables des objets composites. Je souligne le "plus ou moins", car il existe une différence incontournable entre un composite et un objet feuille : le composite doit gérer ses enfants, ce qui signifie probablement qu'il doit avoir une méthode pour récupérer les objets enfants ainsi que les ajouter et les supprimer. Les classes feuilles n'ont évidemment aucun enfant à gérer, puisque c'est dans la nature même des feuilles.

L'approche de ce problème est en grande partie une question de goût. D'un côté, nous pouvons implémenter les objets composites et les feuilles différemment. On pourrait par exemple équiper l'objet composite des méthodes `add_child` et `remove_child` (ou ses équivalents en utilisant la syntaxe des tableaux) et les omettre dans les classes feuilles. La logique sous-jacente est que les objets feuilles n'ont pas d'enfant et n'ont pas besoin de la mécanique de gestion des enfants.

De l'autre côté, notre objectif principal avec le pattern Composite est de rendre les objets feuilles et composites indiscernables. Si le code qui utilise votre composite doit savoir qu'une partie des composants – les composites – expose les méthodes `get_child` et `add_child` tandis que les objets feuilles ne le font pas, alors, les objets composites et feuilles ne sont pas les mêmes. Et si l'on rajoutait les méthodes de gestion des enfants dans un objet feuille ? Que se passerait-il si quelqu'un les appelait ? Un appel à `remove_child` n'est pas très grave car les objets feuilles n'ont pas d'enfant et il n'y aura rien à supprimer. Mais si l'on appelait `add_child` sur un objet feuille ? Ignorer l'appel ? Lever une exception ? Aucune des réponses n'est parfaite.

Je répète, cette décision est principalement une question de goût : concevoir des classes feuilles et composites différentes ou charger les classes feuilles avec des méthodes embarrassantes qu'elles ne savent pas gérer. En ce qui me concerne, je préfère ne pas inclure ces méthodes dans les feuilles. Les objets feuilles ne peuvent pas gérer des objets enfants et il faut l'admettre.

Pointeurs dans les deux sens

Jusqu'alors, nous avons considéré le pattern Composite comme une solution strictement orientée "du haut vers le bas". Les objets composites conservent une référence vers leurs sous-objets, mais les composants enfants n'ont aucune information sur leurs parents. Il est donc facile de parcourir l'arbre de la racine vers les feuilles, mais l'inverse est très difficile.

Pour grimper vers le sommet de l'arbre nous devons ajouter dans chaque participant du pattern Composite une référence vers son parent. Le meilleur endroit pour ce code reste la classe composant. Ainsi, le code pour gérer le parent peut être centralisé :

```
class Task
  attr_accessor :name, :parent
  def initialize(name)
    @name = name
    @parent = nil
  end

  def get_time_required
    0.0
  end
end
```

Vu que la relation parent-enfant est gérée dans la classe composite, c'est l'endroit logique où affecter la valeur à l'attribut parent :

```
class CompositeTask < Task
  def initialize(name)
    super(name)
    @sub_tasks = []
  end

  def add_sub_task(task)
    @sub_tasks << task
    task.parent = self
  end

  def remove_sub_task(task)
    @sub_tasks.delete(task)
    task.parent = nil
  end
```

```
  def get_time_required
    time=0.0
    @sub_tasks.each { |task| time += task.get_time_required}
    time
  end
end
```

Les références vers les parents permettent de tracer le chemin de chaque composant vers le parent initial :

```
while task
  puts("task: #{task}")
  task = task.parent
end
```

User et abuser du pattern Composite

La bonne nouvelle concernant le pattern Composite, c'est qu'il n'existe qu'une seule erreur classique dans son implémentation, la mauvaise nouvelle, c'est que les gens la font très souvent. La faute qui survient si fréquemment dans le pattern Composite est la supposition que l'arbre n'a qu'un unique niveau de profondeur, c'est-à-dire que tous les composants enfants de l'objet composite sont des objets feuilles plutôt que d'autre composites. Pour illustrer ce faux pas, imaginez que nous ayons besoin de connaître le nombre des objets feuilles impliqués dans la préparation de gâteaux. On pourrait simplement ajouter le code suivant dans la classe `CompositeTask` :

```
#
# La mauvaise technique
#
class CompositeTask < Task
  # Beaucoup de code omis...
  def total_number_basic_tasks
    @sub_tasks.length
  end
end
```

C'est faisable, mais ce n'est pas bien. Cette implémentation ignore le fait que chacune de ces sous-tâches pourrait être elle-même un énorme composite avec de nombreuses sous-sous-tâches. La bonne solution dans cette situation est de définir la méthode `total_num_of_tasks` dans la classe composant :

```
class Task
  # Beaucoup de code omis...
  def total_number_basic_tasks
    1
  end
end
```

Ensuite, on surcharge la méthode dans la classe composite avant de parcourir l'arbre récursivement :

```
class CompositeTask < Task
  # Beaucoup de code omis...
  def total_number_basic_tasks
    total = 0
    @sub_tasks.each {|task| total += task.total_number_basic_tasks}
    total
  end
end
```

Souvenez-vous que la puissance du pattern Composite réside dans sa capacité à décrire des arbres de profondeur illimitée. Ne faites donc pas tous ces efforts pour finalement sacrifier cet avantage en écrivant quelques lignes de code mal choisies.

Les composites dans le monde réel

Lorsque l'on recherche des exemples réels du pattern Composite dans la base de code Ruby, les bibliothèques de l'interface graphique utilisateur sautent immédiatement aux yeux. Toutes les interfaces graphiques modernes supportent une palette de composants de base comme les étiquettes, les champs de texte et les menus. Ces composants graphiques de base ont beaucoup de points communs : que ce soit un bouton, une étiquette ou un élément de menu, ils ont tous une police de caractères, une couleur du premier et de l'arrière-plan, et ils occupent tous un certain espace sur l'écran. Évidemment, les vraies interfaces modernes ne représentent pas simplement une collection de composants graphiques. Une vraie interface est une structure hiérarchique : commencez par une étiquette et un champ, positionnez-les d'une certaine façon, puis groupez-les en un seul élément visuel qui servira pour demander à l'utilisateur son prénom. Combinez cet élément d'entrée du prénom avec un élément similaire pour le nom de famille et le numéro de sécurité sociale. Arrangez-les dans un composant graphique encore plus grand et complexe. Si vous avez lu ce chapitre attentivement, le processus doit vous paraître familier : nous venons de développer une interface graphique, un composite.

On peut trouver un bon exemple de l'utilisation de composites dans une bibliothèque d'interfaces graphiques dans FXRuby. FXRuby est une extension Ruby qui amène dans le monde Ruby la bibliothèque des interfaces graphiques open-source et cross-plateforme FOX. FXRuby fournit une grande sélection de widgets d'interface graphique, en commençant par les prosaïques `FXButton` et `FXLabel` jusqu'aux très complexes `FXColorSelector` et `FXTable`. Vous pouvez également construire vos propres objets, aussi complexes que vous voulez, avec `FXHorizontalFrame` et son cousin `FXVerticalFrame`. Ces deux classes jouent le rôle de conteneurs qui permettent d'ajouter d'autres

widgets pour créer un élément d'interface graphique unifié. La différence entre ces deux classes réside dans la manière dont elles affichent leurs sous-éléments : l'une aligne les éléments horizontalement, et l'autre les place verticalement. Que ce soit horizontal ou vertical, les deux classes frames FOX sont des sous-classes de `FXWindow`, tout comme les autres widgets de base.

Par exemple, voici une petite maquette d'un logiciel d'édition de texte à l'aide de FXRuby :

```
require 'rubygems'
require 'fox16'
include Fox
application = FXApp.new("CompositeGUI", "CompositeGUI")
main_window = FXMainWindow.new(application, "Composite",
              ➥ nil, nil, DECOR_ALL)
main_window.width = 400
main_window.height = 200
super_frame = FXVerticalFrame.new(main_window,
              ➥ LAYOUT_FILL_X | LAYOUT_FILL_Y)
FXLabel.new(super_frame, "Text Editor Application")
text_editor = FXHorizontalFrame.new(super_frame,
              ➥ LAYOUT_FILL_X | LAYOUT_FILL_Y)
text = FXText.new(text_editor, nil, 0, TEXT_READONLY | TEXT_WORDWRAP |
              ➥ LAYOUT_FILL_X | LAYOUT_FILL_Y)
text.text = "This is some text."
# La barre de boutons
button_frame = FXVerticalFrame.new(text_editor,
              ➥ LAYOUT_SIDE_RIGHT | LAYOUT_FILL_Y)
FXButton.new(button_frame, "Cut")
FXButton.new(button_frame, "Copy")
FXButton.new(button_frame, "Paste")
application.create
main_window.show (PLACEMENT_SCREEN)
application.run
```

La totalité de l'interface est une série d'objets composites imbriqués. Au premier niveau de l'arbre se trouve `FXMainWindow`, qui contient exactement un élément enfant, un cadre vertical. Ce cadre a un autre enfant ; un cadre horizontal, qui à son tour a... Vous avez sans doute compris le schéma. Sinon consultez la Figure 6.3, qui est un bel exemple du pattern Composite que vous pouvez voir à l'œuvre sur vos écrans.

En conclusion

Une fois la nature récursive de ce pattern assimilée, le pattern Composite devient très simple. Parfois, nous avons besoin de modéliser des objets qui peuvent se regrouper naturellement en composants plus grands. Les objets plus complexes s'inscrivent dans

le pattern Composite s'ils partagent certaines caractéristiques de leurs composants individuels : l'ensemble ressemble à une de ses parties.

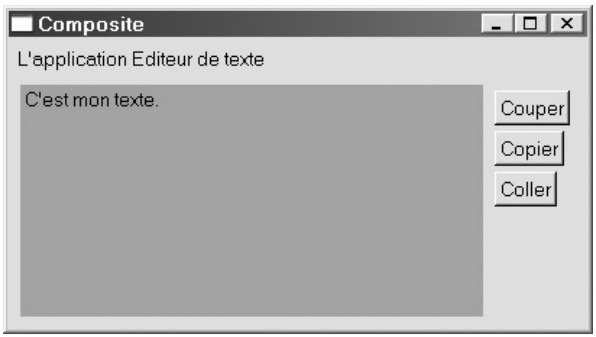


Figure 6.3
Une maquette d'éditeur de texte développé avec FXRuby

Le pattern Composite permet de développer des arbres d'objets à profondeur illimitée dans lesquels on peut gérer chacun des nœuds intérieurs – les composites – de la même façon que les feuilles.

Le pattern Composite est si fondamental qu'il n'est guère surprenant de le voir réapparaître, parfois bien caché, dans d'autres patterns. Comme nous le verrons au Chapitre 15, le pattern Interpréteur n'est qu'une version spécialisée de Composite.

Enfin, il est difficile d'imaginer le pattern Composite sans le pattern Itérateur. La raison pour laquelle les deux sont inséparables va vous apparaître très vite car le pattern Itérateur fait l'objet du chapitre suivant.

7

Accéder à une collection avec l'Itérateur

Au Chapitre 6, nous avons examiné les composites – des objets qui ressemblent à de simples composants, mais qui sont en vérité composés eux-mêmes d'une collection de sous-composants. Évidemment, un objet n'a pas besoin d'être un composite pour avoir connaissance d'une collection d'autres objets. Un objet `Employee` peut avoir une collection de membres de sa famille, ou de numéros de téléphone, ou, dans le cas d'un cadre bien payé, des adresses de plusieurs résidences luxueuses. Dans cette situation, il serait pratique de pouvoir accéder à ces sous-objets séquentiellement sans pour autant connaître les détails du mode de stockage utilisé par le conteneur.

Dans ce chapitre, nous allons explorer le pattern Itérateur, une technique qui permet à un objet conteneur d'ouvrir l'accès à sa collection de sous-objets. Nous verrons les deux types d'itérateurs basiques et nous aurons enfin une explication concernant ces étranges boucles `each`, que l'on rencontre fréquemment dans Ruby.

Itérateurs externes

Voici comment les membres du GoF formulent le but du pattern Itérateur :

> *Fournir un moyen d'accéder de façon séquentielle aux éléments d'un objet collection sans exposer sa représentation interne.*

Formulé autrement, le pattern Itérateur propose au monde extérieur un pointeur mobile vers des objets stockés dans un conteneur de collection opaque.

Si vous êtes un programmeur Java vous connaissez probablement l'interface `java.util.Iterator` et son frère aîné `java.util.Enumeration`.

Voici un exemple d'utilisation typique d'un itérateur Java :

```
ArrayList list = new ArrayList();
list.add("red");
list.add("green");
list.add("blue");
for( Iterator i = list.iterator(); i.hasNext();) {
  System.out.println( "item: " + i.next());
}
```

On trouve également des itérateurs dans les endroits inattendus. Par exemple, on peut voir `java.util.StringTokenizer` comme un itérateur qui nous permet de parcourir tous les tokens d'une chaîne de caractères. De la même manière, JDBC propose la classe `ResultSet`, qui nous permet d'itérer sur chaque ligne du résultat renvoyé pour une requête SQL.

Ce style d'itérateur est souvent appelé l'itérateur externe. "Externe" indique que l'itérateur est un objet séparé du conteneur de collection. Nous verrons bientôt que ce n'est pas le seul itérateur sur la liste. Mais voyons d'abord à quoi ressemblerait un itérateur externe en Ruby.

Il est assez facile de construire des itérateurs externes à la Java en Ruby. Une implémentation simple, quoique trop limitée pour l'usage réel, pourrait ressembler à ceci :

```
class ArrayIterator
  def initialize(array)
    @array = array
    @index = 0
  end

  def has_next?
    @index < @array.length
  end

  def item
    @array[@index]
  end

  def next_item
    value = @array[@index]
    @index += 1
    value
  end
end
```

`ArrayIterator` est une traduction directe de l'itérateur Java en Ruby, avec un ajout : la méthode pour retourner l'élément courant (étrangement, cette fonction est absente dans l'implémentation Java). Voici une façon d'appliquer notre nouvel itérateur :

```
array = ['red', 'green', 'blue']
i = ArrayIterator.new(array)
```

```
while i.has_next?
  puts("item: #{i.next_item}")
end
```

L'exécution de ce code provoque le résultat attendu :

```
item: red
item: green
item: blue
```

En quelques lignes de code notre `ArrayIterator` nous fournit tout ce qui est nécessaire pour itérer sur n'importe quel tableau Ruby. En prime, grâce au typage dynamique et flexible de Ruby, `ArrayIterator` fonctionne avec toute classe qui expose la méthode `length` et qui peut être indexée par un entier. `String` fait partie de ces classes, notre `ArrayIterator` conviendrait donc très bien aux chaînes de caractères :

```
i = ArrayIterator.new('abc')
while i.has_next?
  puts("item: #{i.next_item.chr}")
end
```

Exécutez ce code et vous obtiendrez le résultat suivant :

```
item: a
item: b
item: c
```

Le seul souci lorsqu'on utilise `ArrayIterator` avec des chaînes, c'est que la méthode `[n]` de la classe `String` renvoie le caractère `n` sous forme d'un nombre, son code ASCII, d'où le besoin d'appeler la méthode `chr` dans l'exemple ci-dessus.

Étant donné la facilité de développement d'`ArrayIterator`, il est étonnant que les itérateurs externes soient si rares en Ruby. Il s'avère que Ruby propose quelque chose de mieux, et ce quelque chose est fondé sur nos vieux amis, les blocs de code et l'objet `Proc`.

Itérateurs internes

En y réfléchissant bien, le but d'un itérateur est d'amener votre code vers chaque sous-objet d'un conteneur. La solution des itérateurs externes traditionnels revient à fournir une longue gaffe, l'objet itérateur, qui peut être utilisé pour récupérer des sous-objets de la collection sans plonger dans les détails de l'objet conteneur. Mais avec un bloc de code on peut passer la logique métier à l'objet conteneur. Ensuite, cet objet appellera ce code pour chacun de ses sous-objets. Vu que toutes les actions d'itération se déroulent à l'intérieur de l'objet conteneur, les itérateurs fondés sur des blocs de code sont nommés des itérateurs internes.

Développer un itérateur interne pour des tableaux est très simple. Il suffit de définir une méthode qui appelle le bloc de code (à l'aide de `yield`) pour chaque élément[1] :

```
def for_each_element(array)
  i = 0
  while i < array.length
    yield(array[i])
    i += 1
  end
end
```

Pour utiliser notre itérateur interne on ajoute un bloc de code à la fin de l'appel de méthode :

```
a = [10, 20, 30]
for_each_element(a) {|element| puts("L'élément est #{element}")}
```

Il s'avère que nous n'avons même pas besoin de `for_each_element` car la classe `Array` propose un appel de méthode itérateur `each`. Tout comme notre méthode `for_each_element`, `each` accepte un bloc de code en paramètre et appelle ce bloc pour chaque élément du tableau :

```
a.each {|element| puts("L'élément est #{element}")}
```

Exécutez une des deux versions du code précédent et vous obtiendrez le résultat suivant :

```
L'élément est 10
L'élément est 20
L'élément est 30
```

La méthode `each` explique toutes ces étranges boucles `each` que vous voyez partout dans ce livre. Ces boucles sont non pas de vraies boucles incorporées dans le langage mais plutôt des itérateurs internes.

Itérateurs internes versus itérateurs externes

Bien que les itérateurs internes et externes fassent essentiellement le même travail en parcourant les éléments d'une collection, il faut tenir compte de certaines différences d'ordre pratique. Les itérateurs externes ont certainement leurs avantages. Par exemple, dans le cas d'itérateur externe l'itération est actionnée par le code client. Avec l'itérateur externe vous n'appelez `next` que lorsque vous êtes prêt pour traiter l'élément suivant. Dans le cas d'un itérateur interne, l'objet conteneur force sans cesse le bloc de code à recevoir un élément après l'autre.

---

1. Un vrai programme Ruby ajouterait probablement la méthode `for_each_element` à la classe `String`, en Ruby c'est très facile à réaliser. Pour en savoir plus, voyez le Chapitre 9.

Dans la plupart des cas, cette différence est négligeable. Mais que se passe-t-il si vous essayez de fusionner le contenu de deux tableaux ordonnés dans un seul tableau ordonné ? Ce type d'opération est assez simple avec un itérateur externe tel qu'`ArrayIterator` : on crée un itérateur pour les deux tableaux d'entrée et on boucle en récupérant la valeur inférieure dans un des itérateurs. Cette valeur est ensuite placée dans le tableau de sortie :

```
def merge(array1, array2)
  merged = []
  iterator1 = ArrayIterator.new(array1)
  iterator2 = ArrayIterator.new(array2)
  while( iterator1.has_next? and iterator2.has_next? )
    if iterator1.item < iterator2.item
      merged << iterator1.next_item
    else
      merged << iterator2.next_item
    end
  end
  # Charger les éléments restants du tableau array1
  while( iterator1.has_next?)
    merged << iterator1.next_item
  end
  # Charger les éléments restants du tableau array2
  while( iterator2.has_next?)
    merged << iterator2.next_item
  end
  merged
end
```

Je ne suis pas sûr de bien voir comment implémenter cette même fonction avec des itérateurs internes.

Le deuxième avantage des itérateurs externes réside dans leur position. Puisqu'ils sont externes, ils sont partageables, on peut les passer dans d'autres méthodes et objets. Bien entendu, c'est une épée à double tranchant : vous obtenez la flexibilité, mais vous devez être sûr de votre code. Il faut faire particulièrement attention aux multiples threads qui accèdent à un itérateur externe sans précaution concernant les accès concurrents.

Les atouts principaux des itérateurs internes sont la simplicité et la clarté du code. Les itérateurs externes ont un élément de plus, l'objet itérateur. Dans notre exemple, avec les tableaux nous n'avons pas seulement le tableau et le code client, mais aussi l'objet `ArrayIterator` indépendant. Avec les itérateurs internes il n'y a aucun objet itérateur à gérer ("Est-ce que j'ai déjà appelé `next` ?"), juste quelques lignes de code regroupées dans un bloc.

L'incomparable Enumerable

Si vous créez une classe conteneur équipée d'un itérateur interne, vous devriez probablement envisager l'inclusion d'un module mixin appelé `Enumerable`. `Enumerable` est comme une publicité pour un gadget : pour ajouter `Enumerable`, il suffit de s'assurer que votre méthode d'itérateur interne se nomme `each` et que les éléments sur lesquels vous itérez possèdent une implémentation raisonnable de l'opérateur de comparaison `<=>`. Pour ce prix modique `Enumerable` ajoute à votre classe toute une gamme de méthodes pratiques. Le mixin `Enumerable` vous offre une panoplie de méthodes très commodes comme, par exemple, `include?(obj)`, qui retourne `true` si l'objet fourni en paramètre fait partie de votre collection, ou encore `min` et `max`, qui retournent exactement ce que l'on attend d'eux.

Le mixin `Enumerable` vous fournit aussi des méthodes plus exotiques comme `all?`, qui accepte un bloc et renvoie `true` si le bloc s'évalue à `true` pour tous les éléments. Comme la classe `Array` inclut le mixin `Enumerable`, nous pouvons écrire une ligne de code très simple qui renverra `true` si la longueur de chaque chaîne du tableau est inférieure à quatre caractères :

```
a = [ 'joe', 'sam', 'george' ]
a.all? { element| element.length < 4}
```

La chaîne `'george'` est plus longue que quatre caractères, donc l'appel à `all?` dans cet exemple renvoie `false`. `any?` est une méthode semblable à `all?`. Elle retourne `true` si le bloc renvoie `true` pour au moins un des éléments de l'itérateur. Vu que la longueur de `'joe'` et `'sam'` est inférieure à quatre caractères, le code suivant renvoie `true` :

```
a.any? {|element| element.length < 4}
```

Enfin, `Enumerable` ajoute à votre classe la méthode `sort`, qui ordonne et renvoie tous les éléments du tableau.

Pour comprendre la simplicité avec laquelle on peut ajouter toutes ces fonctionnalités à nos propres classes, imaginez que vous ayez deux classes : une première qui modélise un compte financier et une seconde qui gère un portefeuille de comptes :

```
class Account
  attr_accessor :name, :balance
  def initialize(name, balance)
    @name = name
    @balance = balance
  end

  def <=>(other)
    balance <=> other.balance
  end
end
```

```
class Portfolio
  include Enumerable
  def initialize
    @accounts = []
  end

  def each(&bloc)
    @accounts.each(&bloc)
  end

  def add_account (account)
    @accounts << account
  end
end
```

Nous incluons simplement le module mixin `Enumerable` dans `Portfolio` et définissons la méthode `each`, et voilà `Portfolio` équipé de toute la panoplie des méthodes `Enumerable`. Par exemple, nous pouvons désormais savoir de façon très simple si au moins un des comptes dans le portefeuille contient 2 000 dollars ou plus :

```
my_portfolio.any? { account| account.balance > 2000}
```

Nous pouvons également savoir si tous les comptes contiennent au moins 10 euros :

```
my_portfolio.all? {|account| account.balance > = 10}
```

User et abuser du pattern Itérateur

Itérateur figure parmi les patterns les plus répandus et les plus pratiques mais il présente néanmoins quelques épines prêtes à piquer l'imprudent. Le danger principal est le suivant : que se passe-t-il si l'objet conteneur change lorsque vous êtes en train d'itérer dessus ?

Imaginez que vous parcouriez une liste et que juste avant d'arriver au troisième élément quelqu'un supprime cet élément de la liste. Quel serait le résultat ? Est-ce que l'itérateur doit vous présenter l'objet non existant ? Ou continuer vers le quatrième élément comme si de rien n'était ? Ou peut-être lancer une exception ?

Malheureusement, aucun des itérateurs développés dans ce chapitre ne réagit particulièrement bien à ce genre de changement. Souvenez-vous que notre `ArrayIterator` externe fonctionnait en conservant l'indice de l'élément courant. La suppression des éléments que nous n'avons pas encore vus ne pose aucun problème, néanmoins, toute modification au début du tableau provoquera immanquablement des ravages dans l'indexation.

Créer une copie du tableau parcouru dans le constructeur de l'objet itérateur peut rendre `ArrayIterator` résistant aux changements opérés sur ce tableau :

```
class ChangeResistantArrayIterator
def initialize(array)
  @array = Array.new(array)
  @index = 0
end
...
```

Le nouvel itérateur crée une copie incomplète (shallow copy) du tableau – la copie pointe vers le contenu d'origine, qui n'est pas copié – et parcourt le nouveau tableau. Grâce à ce nouvel itérateur nous obtenons une capture du tableau, résistante aux changements, sur laquelle nous pouvons itérer.

Les itérateurs internes sont sensibles au même problème de modification concurrente que les itérateurs externes. Par exemple, c'est probablement une très mauvaise idée de faire ceci :

```
array=['red', 'green', 'blue', 'purple']
array.each do color
  puts(color)
  if color == 'green'
    array.delete(color)
  end
end
```

Ce code affiche

```
red
green
purple
```

En supprimant l'entrée `'green'` nous avons réussi à semer la pagaille dans l'indexation, le résultat étant que l'entrée `'blue'` n'est pas affichée.

Les itérateurs internes peuvent également opérer sur une copie indépendante de l'objet conteneur pour éviter tout risque de modification en cours d'itération comme ce que nous avons fait dans la classe `ChangeResistantArrayIterator`. L'implémentation pourrait ressembler au code suivant :

```
def change_resistant_for_each_element(array)
  copy = Array.new(array)
  i = 0
  while i < copy.length
    yield(copy[i])
    i += 1
  end
end
```

Pour résumer, un programme à plusieurs fils d'exécution (threads) est un milieu particulièrement dangereux pour des itérateurs. Il faut prendre toutes les précautions habituelles pour s'assurer qu'un thread ne tire pas le tapis sous les pieds de votre itérateur.

Les itérateurs dans le monde réel

Les itérateurs – internes pour la plupart mais occasionnellement externes – sont si fréquents en Ruby qu'il est difficile de choisir par quel exemple commencer. En fait, les tableaux Ruby ont deux autres itérateurs internes en plus de `each`. La méthode `reverse_each` boucle sur des éléments du tableau en commençant par le dernier pour remonter jusqu'au premier, alors que `each_index` appelle le bloc passé en argument sur chacun des indices du tableau au lieu de ses éléments.

La classe `String` possède une méthode `each` qui parcourt chaque ligne de la chaîne (oui, chaque ligne et non pas chaque caractère) ainsi que la méthode `each_byte`. Les objets `String` ont également la formidable méthode `scan`, qui accepte en paramètre une expression régulière et itère sur chaque occurrence trouvée dans la chaîne. Par exemple, nous pouvons ainsi rechercher des mots commençant par la lettre `'p'` :

```
s = 'Peter Piper picked a peck of pickled peppers'
s.scan(/[Pp]\w*/) { |word| puts("Le mot est #{word}")}
```

Ce code affiche beaucoup de mots en `'p'` :

```
Le mot est Peter
Le mot est Piper
Le mot est picked
Le mot est peck
Le mot est pickled
Le mot est peppers
```

Vous ne serez pas étonné d'apprendre que la classe `Hash` supporte elle aussi une riche palette d'itérateurs. Nous avons `each_key`, qui appelle un bloc de code pour chaque clé du tableau associatif :

```
h = {'nom' => 'russ', 'yeux' => 'bleu', 'sexe' => 'mâle'}
h.each_key {|key| puts(key)}
```

Ce code affiche le résultat suivant :

```
nom
sexe
yeux
```

La classe `Hash` offre également la méthode `each_value` :

```
h.each_value { |value| puts(value)}
```

Ce code affiche le résultat suivant :

```
russ
mâle
bleu
```

Enfin, la méthode `each` classique est aussi disponible :

```
h.each {|key, value| puts("#{key} #{value}")}
```

La méthode `each` itère sur des paires clé/valeur du tableau associatif, le code affiche donc le résultat suivant :

```
nom russ
sexe mâle
yeux bleu
```

Des itérateurs externes en Ruby sont plus difficiles à trouver. L'objet `IO` constitue un spécimen intéressant. La classe `IO` est une classe chargée de gérer des flux d'entrée-sortie. C'est un objet amphibie à l'implémentation élégante : il propose au choix un itérateur externe ou interne. On peut ouvrir un fichier et lire chaque ligne avec un itérateur externe de façon classique :

```
f = File.open('names.txt')
while not f.eof?
  puts(f.readline)
end
f.close
```

L'objet `IO` expose également la méthode `each` (aussi nommée `each_line`), qui implémente un itérateur interne renvoyant chaque ligne d'un fichier :

```
f = File.open('names.txt')
f.each {|line| puts(line)}
f.close
```

Les fichiers non orientés ligne peuvent être traités par la méthode d'itération `each_byte` :

```
f.each_byte {|byte| puts(byte)}
```

Si vos programmes font beaucoup d'opérations d'entrée et sortie, vous devriez probablement vous intéresser de près à la classe `Pathname`. `Pathname` ambitionne de vous offrir tous les outils nécessaires à la manipulation des dossiers et des chemins d'un système de fichiers. On crée un objet `Pathname` en passant au constructeur le chemin concerné :

```
pn = Pathname.new('/usr/local/lib/ruby/1.8')
```

En complément d'un éventail de méthodes qui n'ont aucune relation avec des itérateurs, `Pathname` fournit l'itérateur `each_filename`, qui boucle sur des composants du chemin passé en argument.

Si vous exécutez le code suivant :

```
pn.each_filename {|file| puts("File: #{file}")}
```

vous obtiendrez :

```
File: usr
File: local
File: lib
File: ruby
File: 1.8
```

Vous pouvez également changer de dimension : la méthode `each_entry` itère sur le contenu du dossier vers lequel pointe l'objet `Pathname`. Donc si vous exécutez

```
pn.each_entry { entry| puts("Entry: #{entry}")}
```

vous verrez le contenu de `/usr/local/lib/ruby/1.8`[1] :

```
Entry: .
Entry: ..
Entry: i686-linux
Entry: shellwords.rb
Entry: mailread.rb
...
```

Enfin, mon itérateur interne préféré est celui proposé par le module `ObjectSpace`. `ObjectSpace` ouvre une fenêtre vers l'univers des objets présents dans votre interpréteur Ruby. L'itérateur fondamental fourni par `ObjectSpace` est la méthode `each_object`. Il itère sur tous les objets Ruby, c'est-à-dire tout ce qui est chargé dans votre interpréteur Ruby :

```
ObjectSpace.each_object { object| puts("Object: #{object}")}
```

La méthode `each_object` accepte un argument facultatif qui peut être une classe ou un module. Si l'argument est présent, `each_object` itère uniquement sur les instances de cette classe ou de ce module. Eh oui, les sous-classes sont incluses ! Si je voulais afficher tous les nombres connus de mon interpréteur Ruby, je pourrais faire ceci :

```
ObjectSpace.each_object(Numeric) {|n| puts("The number is #{n}")}
```

La capacité d'introspection de Ruby est assez sensationnelle. Avec `ObjectSpace` vous pouvez implémenter votre propre système d'inspection de la mémoire : il suffit de créer un thread qui observe les objets intéressants tout en affichant un rapport en conséquence. Par exemple, une classe pourrait utiliser `ObjectSpace` pour retrouver toutes les

1. Vous verrez ce résultat si vous utilisez un système d'exploitation UNIX et si Ruby est installé dans `/usr/local/lib`.

instances d'elle-même. Rails recourt à `ObjectSpace` pour construire la méthode qui retrouve toutes les sous-classes d'une classe donnée :

```
def subclasses_of(superclass)
  subclasses = []
  ObjectSpace.each_object(Class) do |k|
    next if !k.ancestors.include?(superclass) || superclass == k ||
    ➥ to_s.include?('::') || subclasses.include?(k.to_s)
    subclasses << k.to_s
  end
  subclasses
end
```

Si l'on exécute

```
subclasses_of(Numeric)
```

on recevra un tableau qui contient `"Bignum"`, `"Float"`, `"Fixnum"` et `"Integer"`. Je le répète, les capacités d'introspection de Ruby sont vraiment sensationnelles.

En conclusion

Dans ce chapitre nous avons exploré deux formes fondamentales d'itérateurs. La première version, et probablement la plus courante, est l'itérateur externe : un objet qui pointe vers un élément d'une collection. Dans le cas d'un itérateur interne, au lieu de passer une sorte de pointeur, nous descendons le code de gestion des éléments vers les sous-objets.

Nous avons également rencontré le module `Enumerable`, qui peut améliorer les possibilités d'itération sur tout type de collection. Nous avons par ailleurs jeté un coup d'œil sur le côté obscur des itérateurs, quand notre collection peut être modifiée pendant que le processus d'itération est en cours. Enfin, nous avons effectué une visite guidée avec `ObjectSpace`, qui peut parcourir les entrailles de l'interpréteur Ruby et nous montrer des choses que l'on ne pensait jamais voir.

Les itérateurs Ruby sont un excellent exemple de la beauté du langage. Ruby tire parti de la flexibilité des objets `Proc`, des blocs de code et des itérateurs internes plutôt que fournir des itérateurs externes spécialisés pour chaque classe conteneur. Les itérateurs internes sont très faciles à écrire – vous créez une seule méthode au lieu d'une toute nouvelle classe –, Ruby encourage donc les programmeurs à choisir un itérateur optimal pour leurs besoins. La puissance de cette approche devient évidente avec la grande collection des itérateurs disponibles dans la bibliothèque standard de Ruby, où l'on peut obtenir tout de `each_byte` dans une chaîne de caractères à `each_object` dans l'interpréteur Ruby lui-même.

8

Effectuer des actions avec Command

J'ai mentionné au Chapitre 1 que lorsque j'étais lycéen et à mes débuts à l'université je passais un temps considérable à travailler dans un commerce local. Ce petit magasin essayait avec beaucoup de difficultés de concurrencer des grands supermarchés et offrait donc des services introuvables au mégamarché local. Les clients pouvaient notamment nous appeler et dicter au téléphone leur liste de courses. Nous étions plus que contents de rassembler les haricots, le beurre et le saucisson et de les livrer gratuitement. Certains de nos clients avaient même des commandes permanentes. Ils appelaient et demandaient qu'on leur livre leur commande habituelle. C'était donc à moi de préparer la livraison avec la liste des courses à la main.

Les listes des courses de ma jeunesse ressemblent beaucoup aux commandes qui ont donné leur nom au pattern Command qui nous occupe dans ce chapitre. Tout comme une liste de courses, une commande du pattern Command est une instruction destinée à déclencher une action assez spécifique. Tout comme une liste, une commande peut être exécutée tout de suite ou ultérieurement lorsqu'un événement particulier se produit.

Puisque le pattern Command est un des patterns les plus polyvalents couverts dans ce livre, la discussion qui suit va plutôt ressembler à une revue d'ensemble. On commencera par l'usage très répandu du pattern Command dans les interfaces utilisateur graphiques, puis nous verrons comment enregistrer les actions effectuées ainsi que les actions qui restent à faire. Enfin, nous utiliserons le pattern Command pour annuler des actions déjà exécutées ou, *a contrario*, défaire ce qui vient d'être fait.

L'explosion de sous-classes

Imaginez que vous soyez en train de développer SlickUI, un nouveau framework d'interface graphique. Vous créez des boutons magnifiques, des boîtes de dialogue exquises et des icônes à tomber à la renverse. Mais, une fois les éléments graphiques de votre framework joliment décorés, vous vous retrouvez face à un problème critique : comment mettre l'interface en action pour qu'elle soit utile à quelque chose ?

Imaginez que votre classe bouton soit conçue pour appeler la méthode `on_button_push` dès que l'utilisateur appuie sur le bouton à l'écran :

```
class SlickButton
  #
  # Beaucoup de code graphique et logique de gestion omis
  # ...
  #
  def on button push
    #
    # Faire quelque chose lorsque le bouton est appuyé
    #
  end
end
```

Que faut-il écrire à l'intérieur de la méthode `on_button_push` ? Vous espérez que SlickUI deviendra extrêmement populaire et qu'il sera utilisé par des milliers de programmeurs partout dans le monde. Ils créeront des millions d'instances de SlickButton. Une équipe de programmeurs développera peut-être un éditeur de texte et elle aura besoin de boutons pour créer de nouveaux documents et pour enregistrer les documents en cours de rédaction. Une autre équipe de projet se concentrera plutôt sur un utilitaire de gestion de réseau et elle aura donc besoin d'un bouton pour ouvrir une connexion réseau. La difficulté dans tout cela, c'est qu'au moment où vous écrivez la classe SlickButton vous n'avez aucune idée de toutes les fonctions que vos futurs clients vont pouvoir accrocher à tous ces boutons.

Une des solutions à ce problème consiste à recourir à notre outil presque universel, mais légèrement discrédité : l'héritage. Vous pourriez demander à vos utilisateurs de sous-classer votre classe pour chaque type de bouton comme ceci :

```
class SaveButton < SlickButton
  def on button push
    #
    # Enregistrer le document courant...
    #
  end
end
```

```
class NewDocumentButton < SlickButton
  def on_button_push
    #
    # Créer un nouveau document...
    #
  end
end
```

Malheureusement, une application avec une interface graphique complexe aura des dizaines ou même des centaines de boutons et, par conséquent, des dizaines ou des centaines de sous-classes de `SlickButton`, sans parler des autres éléments graphiques de l'interface comme les menus et les boutons radio. Qui plus est, l'héritage est permanent. Et si vous vouliez que votre bouton fasse une action avant d'ouvrir la feuille de calcul et une action juste après l'avoir ouvert ? Si vous écrivez une sous-classe de `Button`, soit vous avez besoin de deux sous-classes de `Button` séparées soit vous êtes obligé de coder la logique "Le fichier est-il ouvert ?" dans une seule sous-classe de `Button`. Les deux techniques ne sont pas propres. Existe-t-il un moyen plus simple ?

Un moyen plus simple

La bonne approche à ce problème consiste à encapsuler l'action à exécuter lorsqu'on clique sur un bouton ou sur un élément de menu. Autrement dit, il faut extraire le code de gestion de l'action provoquée par le bouton ou le menu dans un objet séparé qui ne fait qu'attendre son exécution. Lorsqu'il est exécuté, l'action effectue la tâche spécifique à l'application. Ces actions encapsulées sont des commandes du pattern Command.

Pour appliquer le pattern Command à notre exemple il suffit de conserver la commande dans l'objet bouton :

```
class SlickButton
  attr_accessor :command
  def initialize(command)
    @command = command
  end
  #
  # Beaucoup de code métier et graphique omis
  # ...
  #
  def on_button_push
    @command.execute if @command
  end
end
```

Nous pouvons définir des commandes différentes pour toutes les actions possibles de nos boutons :

```
class SaveCommand
  def execute
    #
    # Enregistrer le document courant...
    #
  end
end
```

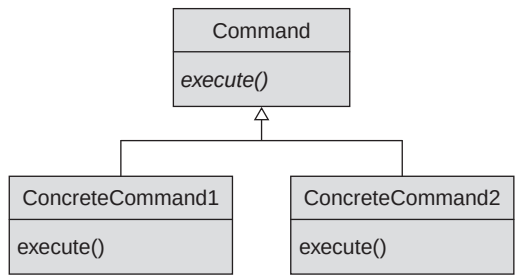
Et nous passons la commande concrète à la création du bouton :

```
save_button = SlickButton.new( SaveCommand.new )
```

Factoriser le code de l'action dans son propre objet est l'idée essentielle du pattern Command. Ce pattern sépare la partie variable, la tâche à accomplir lorsque le bouton est sélectionné, de la partie statique, à savoir la classe de bouton générique apportée par le framework graphique. Puisque la connexion entre le bouton et la commande s'établit au moment de l'exécution – le bouton stocke simplement une référence vers la commande à déclencher lorsqu'il est sélectionné –, il est facile de remplacer les commandes à la volée et de changer le comportement du bouton au moment de l'exécution.

Comme vous pouvez le voir sur le diagramme UML (voir Figure 8.1), la structure du pattern Command est très simple. Elle consiste en un certain nombre de classes qui partagent la même interface.

Figure 8.1
Le pattern Command



Des blocs de code comme commandes

Nous avons vu qu'une commande n'est qu'un objet qui encapsule un fragment de code spécialisé. La seule raison d'exister d'une commande est d'appeler ce code au bon moment. Cette description doit vous être familière : c'est la définition assez précise d'un bloc de code Ruby ou d'un objet `Proc`. Souvenez-vous qu'un objet `Proc` encapsule du code qui attend d'être exécuté.

Le pattern Command se traduit très naturellement en bloc de code. Voici notre classe `SlickButton` restructurée pour employer des blocs de code :

```
class SlickButton
  attr_accessor :command
  def initialize(&bloc)
    @command = bloc
  end
  #
  # Beaucoup de code métier et graphique omis
  # ...
  #
  def on_button_push
    @command.call if @command
  end
end
```

Pour créer notre nouveau `SlickButton` fondé sur des blocs de code nous passons un bloc de code à la création du bouton :

```
new_button = SlickButton.new do
  #
  # Création d'un nouveau document...
  #
end
```

Dans un monde Ruby fait de blocs de code et d'objets `Proc`, les classes codées à la main comme `SaveCommand` sont-elles totalement dépassées ? Pas vraiment. Tout dépend de la complexité de la tâche. Si vous avez besoin d'une commande simple qui exécute quelques actions très claires, je vous recommande vivement d'opter pour un objet `Proc`. En revanche, si votre tâche est relativement complexe (comme le fait de garder beaucoup d'information sur le contexte ou bien de nécessiter plusieurs méthodes), n'hésitez pas à créer une classe de commande spécifique.

Les commandes d'enregistrement

Les boutons et les commandes qui les accompagnent constituent certes un bon exemple du pattern Command mais son usage est loin d'être limité aux interfaces graphiques. Par exemple, le pattern Command se révèle très utile pour garder trace des actions effectuées. Imaginez que vous développiez un utilitaire d'installation de logiciels. Un programme d'installation a typiquement besoin de créer, copier, déplacer et parfois supprimer des fichiers. Il est aussi probable que l'utilisateur souhaite prendre connaissance des actions que l'installeur s'apprête à effectuer avant qu'elles ne soient lancées ou bien savoir ce qu'a fait l'installeur après exécution. Assurer le suivi de ces opérations est facile si les actions à effectuer sont organisées sous forme de commandes.

Les commandes d'installation vont contenir un certain nombre d'informations sur leur état, nous allons donc les coder comme des classes séparées selon le style classique du pattern Command. Commençons par définir quelques commandes de manipulation

de fichiers. Chacune d'elles implémente la méthode `describe` ainsi que la méthode `execute`. Pour commencer, voici la classe parent des commandes :

```
class Command
  attr_reader :description
  def initialize(description)
    @description = description
  end

  def execute
  end
end
```

Ensuite, définissons la commande permettant de créer un fichier et d'écrire son contenu à partir d'une chaîne de caractères :

```
class CreateFile < Command
  def initialize(path, contents)
    super("Create file: #{path}")
    @path = path
    @contents = contents
  end

  def execute
    f = File.open(@path, "w")
    f.write(@contents)
    f.close
  end
end
```

Nous pouvons également avoir besoin d'une commande pour supprimer un fichier :

```
class DeleteFile < Command
  def initialize(path)
    super("Delete file: #{path}")
    @path = path
  end

  def execute
    File.delete(@path)
  end
end
```

Et peut-être d'une commande pour copier un fichier dans un autre :

```
class CopyFile < Command
  def initialize(source, target)
    super("Copy file: #{source} to #{target}")
    @source = source
    @target = target
  end

  def exécute
    FileUtils.copy(@source, @target)
  end
end
```

Évidemment, nous aurions pu étendre davantage les classes de commandes, par exemple pour renommer des fichiers, modifier les droits d'accès ou créer des dossiers, mais arrêtons-nous là pour le moment.

Puisque nous essayons de garder un suivi de ce que nous nous apprêtons à faire ou de ce que nous avons déjà fait, nous avons besoin d'une classe pour rassembler toutes nos commandes. Hum ! une classe qui se comporte comme une commande, mais qui en réalité n'est qu'une façade pour un certain nombre de sous-commandes. Cela ressemble fort à un composite :

```
class CompositeCommand < Command
  def initialize
    @commands = []
  end

  def add_command(cmd)
    @commands << cmd
  end

  def execute
    @commands.each { |cmd| cmd.execute}
  end

  def description
    description = ''
    @commands.each { |cmd| description += cmd.description + "\n"}
    description
  end
end
```

Outre la satisfaction de mettre en œuvre un pattern déjà étudié, `CompositeCommand` nous permet d'informer l'utilisateur de ce que nous faisons exactement avec son système. On pourrait par exemple créer un nouveau fichier, le copier dans un autre fichier, et puis supprimer le premier :

```
cmds = CompositeCommand.new
cmds.add_command(CreateFile.new('file1.txt', "hello world\n"))
cmds.add_command(CopyFile.new('file1.txt', 'file2.txt'))
cmds.add_command(DeleteFile.new('file1.txt'))
```

Pour véritablement exécuter toutes ces manipulations sur les fichiers on appelle simplement :

```
cmds.execute
```

L'avantage majeur de cette technique réside dans sa capacité à expliquer à l'utilisateur ce qui se passe à tout moment – que ce soit avant ou après l'exécution des commandes. Par exemple, le code

```
puts(cmds.description)
```

affiche

```
Create file: file1.txt
Copy file: file1.txt to file2.txt
Delete file: file1.txt
```

Annuler des actions avec Command

Permettre à votre client (que ce soit un utilisateur ou un autre programme) d'annuler des actions déjà effectuées est une demande classique. Aujourd'hui, la fonction d'annulation est une nécessité absolue pour tout éditeur de texte, mais on la retrouve aussi ailleurs. Par exemple, la plupart des bases de données supportent le *rollback* de transactions, ce qui est un autre nom d'une fonction d'annulation. En fait, annuler des actions peut devenir une exigence partout où une série de modifications coûteuses en effort est réalisée par un humain (enclin à l'erreur) ou par un programme.

La façon naïve d'implémenter cette opération consiste à garder en mémoire l'état avant le changement et à restituer cet état si le client décide d'annuler la modification. Le problème de cette approche, c'est que les fichiers texte et les documents modifiés (sans parler des bases de données) peuvent être assez volumineux. Faire une copie complète de l'ensemble avant d'apporter une modification peut très vite devenir assez laid et surtout très coûteux en ressources.

Le pattern Command peut également être utile dans cette situation. Une commande – une encapsulation de code pour faire une action spécifique – pourrait aussi, avec quelques améliorations, défaire une action. L'idée est très simple : chaque commande annulable que nous créons possède deux méthodes. Aux côtés de la méthode habituelle `execute`, qui déclenche une action, nous ajoutons la méthode `unexecute` pour l'annuler. Lorsque l'utilisateur fait des changements nous créons une commande après l'autre en les exécutant immédiatement pour provoquer la modification. Mais ces commandes doivent être stockées dans l'ordre d'exécution quelque part dans une liste. Si l'utilisateur change d'avis et décide d'annuler la modification, nous serons ainsi en mesure de retrouver la dernière commande dans la liste et nous pourrons appeler `unexecute`. Nous pouvons même permettre à l'utilisateur de remonter aussi loin qu'il le souhaite dans l'historique des actions en annulant des modifications une par une.

Refaire la modification, c'est-à-dire la possibilité de changer d'avis une fois de plus et de réappliquer les changements qui viennent d'être annulés, s'inscrit élégamment dans la même veine de conception. Pour refaire les actions il suffit de réexécuter les commandes en commençant par la dernière annulée.

Rendons l'explication un peu plus concrète et retournons à l'exemple de l'installeur. Supposons que la demande ne consiste pas simplement à pouvoir expliquer ce que nous faisons avec le système de l'utilisateur mais que nous devions aussi offrir la possibilité de revenir en arrière si l'utilisateur juge que l'installation est une mauvaise idée. On commence par ajouter la méthode `unexecute` à la commande `CreateFile` :

```
class CreateFile < Command
  def initialize(path, contents)
    super "Create file: #{path}"
    @path = path
    @contents = contents
  end

  def execute
    f = File.open(@path, "w")
    f.write(@contents)
    f.close
  end

  def unexecute
    File.delete(@path)
  end
end
```

La méthode `unexecute` porte bien son nom : elle supprime le fichier créé par la commande `execute`. Ce que la méthode `execute` nous donne, la méthode `unexecute` nous le retire.

Un défi un peu plus sérieux nous attend avec la commande `DeleteCommand` car elle est destructive par nature. Pour annuler l'opération de suppression nous sommes obligés de sauvegarder le contenu du fichier d'origine *avant de le supprimer*[1]. Dans un vrai système, on copierait probablement le contenu du fichier dans un répertoire temporaire, mais pour l'exemple contentons-nous de le stocker dans la mémoire :

```
class DeleteFile < Command
  def initialize(path)
    super "Delete file: #{path}"
    @path = path
  end

  def execute
    if File.exists?(@path)
      @contents = File.read(@path)
```

1. Le lecteur sagace (c'est-à-dire vous) aurait déjà compris que l'action `CreateFile` pourrait aussi être destructive. Il est possible que le fichier que l'on essaie de créer existe déjà et soit écrasé par le nouveau. Dans un système réel nous devons gérer cette possibilité ainsi qu'un tas de questions liées aux droits d'accès et de propriété du fichier. Afin de préserver la simplicité des exemples, je vais ignorer toutes ces questions. Parfois, c'est bien de se contenter d'écrire des exemples.

```
      end
      f = File.delete(@path)
    end

    def unexecute
      if @contents
        f = File.open(@path,"w")
        f.write(@contents)
        f.close
      end
    end
  end
```

L'ajout de la méthode `unexecute` à `CopyFile` soulève les mêmes questions que `DeleteFile`. Avant de faire une copie à l'aide de la méthode `execute` il faudrait vérifier que le fichier cible existe et, si c'est le cas, sauvegarder son contenu. La méthode `unexecute` devra rétablir le contenu du fichier s'il existait ou bien l'effacer s'il n'existait pas auparavant.

Enfin, nous avons besoin d'ajouter la méthode `unexecute` à la classe `CompositeCommand` :

```
class CompositeCommand < Command
  # ...
  def unexecute
    @commands.reverse.each { |cmd| cmd.unexecute }
  end
  # ...
end
```

La méthode `unexecute` est généralement l'inverse de la méthode `execute`, elle annule les sous-commandes. Remarquez que nous appelons la méthode `reverse` sur les tableaux de commandes avant d'itérer dessus car pour annuler les actions il faut commencer par la dernière commande et remonter l'historique vers la première.

Créer des files de commandes

Le pattern Command peut également être utile dans les situations où il faut accumuler plusieurs opérations pour ensuite les exécuter ensemble. C'est le mode de fonctionnement courant des installeurs. Un assistant d'installation typique vous permet de dire "oui, je veux le programme de base, oui, je veux la documentation, mais je ne veux pas de fichiers d'exemples". Lorsque vous configurez l'installeur, il compose une sorte de liste des choses à faire : copier le programme, copier la documentation, etc. À la fin l'assistant vous donne la possibilité de changer d'avis. Les choses ne se font véritablement qu'après avoir cliqué sur le bouton d'installation. Il est évident que cette liste de tâches constitue là encore une liste de commandes.

Une situation similaire se produit si une série d'opérations doit être exécutée et que chacune des opérations, exécutée seule, a un coût de démarrage élevé. Par exemple, une petite éternité est souvent nécessaire pour se connecter à une base de données. Si un bon nombre d'opérations sont à réaliser sur la base de données on se retrouve alors face à un choix désagréable : (1) soit conserver une connexion ouverte constamment et gâcher une ressource précieuse, (2) soit dépenser le temps nécessaire pour ouvrir et fermer la connexion à chaque opération.

Dans ce cas de figure, le pattern Command offre une solution. Au lieu d'exécuter chaque opération comme une tâche indépendante, on accumule les commandes dans une liste. La connexion à la base de données peut être ouverte périodiquement pour exécuter toutes les commandes accumulées, ensuite, la liste est vidée.

User et abuser du pattern Command

Le pattern Command présente une particularité qui tend à provoquer un usage excessif. Aussi concis que puisse être un pattern Command en Ruby,

```
class FileDeleteCommand
  def initialize(path)
    @path = path
  end

  def execute
    File.delete(@path)
  end
end
fdc = FileDeleteCommand.new('foo.dat')
fdc.execute
```

il n'en demeure pas moins que rien n'est plus simple que de faire les choses directement :

```
File.delete('foo.dat')
```

L'une des caractéristiques clés du pattern Command tient à sa capacité à séparer la réflexion de l'action. Lorsque vous utilisez ce pattern, vous ne dites plus "fais ceci". Vous dites plutôt "souviens-toi comment faire ceci" et ensuite, plus tard, "fais l'action que tu as retenue". Même la version légère du pattern Command fondée sur des blocs de code Ruby ajoute un sérieux niveau de complexité en raison de ce processus en deux étapes. Assurez-vous que cette complexité est justifiée avant de dégainer le pattern Command.

Dans l'hypothèse où vous avez vraiment besoin du pattern Command, il faut vous assurer que vous avez bien pensé à tout avant de le faire fonctionner. Réfléchissez bien à toutes les circonstances dans lesquelles votre objet commande peut se trouver tant à

l'étape de création qu'à celle de l'exécution. Oui, le fichier principal est ouvert et l'objet crucial est initialisé lorsque je crée la commande. Est-ce que ce sera encore le cas quand la commande sera exécutée ?

D'habitude, pour des commandes simples, prévoir correctement le contexte de création et d'exécution n'est pas très difficile. En général, il suffit de sauvegarder tous les arguments de l'opération dans l'objet commande. Mais les commandes annulables requièrent une vigilance particulière. Beaucoup d'opérations sont destructives : elles effacent des données. Si vous envisagez de créer une commande annulable il faut trouver un moyen de préserver dans l'objet commande les données susceptibles d'être effacées à l'exécution. Cela vous permettra de restaurer ces données en cas d'annulation.

Le pattern Command dans le monde réel

Comme l'indique l'introduction de ce chapitre, on trouve souvent le pattern Command dans des frameworks d'interface graphique. Les frameworks Tk et FXRuby permettent tous les deux d'associer des commandes fondées sur des blocs de code avec des éléments graphiques tels que boutons et autres entrées de menu. Mais on peut également voir le pattern Command dans d'autres parties de la base de code Ruby.

Migration ActiveRecord

La fonctionnalité des migrations d'`ActiveRecord`[1] est dotée d'une implémentation classique du pattern Command annulable. Les migrations ActiveRecord permettent au programmeur de définir le schéma de sa base de données indépendamment du moteur de base de données utilisé. Ce code est écrit en Ruby, naturellement.

Les migrations Rails constituent un exemple tout à fait pertinent pour ce chapitre car chaque partie du schéma est organisée comme une commande. La migration ci-après, par exemple, crée dans la base de données la table `books` :

```
class CreateBookTable < ActiveRecord::Migration
  def self.up
    create_table :books do |t|
      t.column :title, :string
      t.column :author, :string
    end
  end

  def self.down
    drop_table :books
  end
end
```

1. Comme vous le savez, `ActiveRecord` est l'interface des bases de données utilisée par Rails.

Notez que le code de création figure dans la méthode `up`. La méthode `up` est équivalente à la méthode `execute`, elle s'occupe du travail de création de la table `books`. Notre migration propose également la méthode `down`, qui annule l'effet de la commande – la migration – et supprime la table créée par la méthode `up`.

Une application Rails typique définit toute une liste des classes de migrations telles que la classe ci-dessus. Elles ajoutent des classes au fur et à mesure des évolutions de la base de données. La beauté des migrations tient au fait qu'il est possible d'avancer le schéma à l'étape suivante ou de retourner à l'état précédent à l'aide des méthodes `up` et `down`.

Madeleine

Un autre exemple remarquable du pattern Command dans du vrai code Ruby se trouve dans Madeleine. Madeleine est une implémentation Ruby de Prevayler, un projet qui a ses racines dans le monde Java, mais qui s'est répandu dans de nombreux autres langages.

Madeleine est un framework transactionnel à haute performance pour la persistance d'objets qui n'a pas besoin de mappage objet relationnel pour la simple raison qu'il n'utilise pas de base de données relationnelle. Madeleine se fie au package Ruby `Marshal` : un module Ruby qui permet de convertir des objets Ruby en octets et inversement, les octets en objets. Malheureusement, la sériation de vos objets dans des fichiers n'est pas une solution complète à la persistance d'applications. Imaginez la lenteur de votre système si vous deviez réécrire l'affectation des sièges pour l'ensemble des avions d'une compagnie aérienne à chaque fois que quelqu'un souhaite changer de place dans un avion.

Les choses se passeraient bien plus rapidement si l'on pouvait n'enregistrer que les changements : sauvegarder l'état initial de vos objets, et puis écrire les modifications au fur et à mesure de ses évolutions. Attendez, cela a l'air étrangement familier...

Pour avoir une idée de Madeleine, développons un système simple de gestion des ressources humaines. Commençons par l'omniprésente classe `Employee` :

```
require 'rubygems'
require 'madeleine'
class Employee
  attr_accessor :name, :number, :address
  def initialize(name, number, address)
    @name = name
    @number = number
    @address = address
  end

  def to_s
    "Employee: name: #{name} num: #{number} addr: #{address}"
  end
end
```

Ensuite, nous allons créer une classe de gestion des employés. Cette classe conserve un tableau associatif des employés en utilisant leurs numéros comme clés. La classe `EmployeeManager` nous permet d'ajouter un employé, de supprimer un employé existant, de modifier son adresse et de retrouver son numéro :

```
class EmployeeManager
  def initialize
    @employees = {}
  end

  def add_employee(e)
    @employees[e.number] = e
  end

  def change_address(number, address)
    employee = @employees[number]
    raise "No such employee" if not employee
    employee.address = address
  end

  def delete_employee(number)
    @employees.remove(number)
  end

  def find_employee(number)
    @employees[number]
  end
end
```

Rien de très sensationnel pour le moment, mais l'intrigue va s'épaissir. Définissons un ensemble des commandes, une pour chaque opération supportée par la classe `EmployeeManager`. Au début, nous avons `AddEmployee` : la commande pour insérer un nouvel `Employee` dans le tableau associatif `EmployeeManager`. Tout comme les autres commandes, la classe `AddEmployee` consiste en une méthode `initialize` qui stocke suffisamment d'informations pour pouvoir répéter la commande et la méthode `execute` qui exécute véritablement la commande :

```
class AddEmployee
  def initialize(employee)
    @employee = employee
  end

  def execute(system)
    system.add_employee(@employee)
  end
end
```

Les commandes de suppression, de changement d'adresse et de recherche sont très similaires :

```
class DeleteEmployee
  def initialize(number)
    @number = number
  end
```

```
  def execute(system)
    system.delete_employee(@number)
  end
end

class ChangeAddress
  def initialize(number, address)
    @number = number
    @address = address
  end

  def execute(system)
    system.change_address(@number, @address)
  end
end

class FindEmployee
  def initialize(number)
    @number = number
  end

  def execute(system)
    system.find_employee(@number)
  end
end
```

Venons-en maintenant à la partie intéressante : créons un nouvel objet Madeleine en lui passant le nom du dossier où sauvegarder nos données ainsi qu'un bloc de code pour créer une nouvelle instance d'`EmployeeManager :`

```
store = SnapshotMadeleine.new('employees') { EmployeeManager.new }
```

Nous avons également besoin d'un thread pour sauvegarder à intervalles réguliers l'état des objets stockés. Notre thread demande que Madeleine enregistre l'état du système sur le disque toutes les 20 secondes :

```
Thread.new do
  while true
    sleep(20)
      madeleine.take_snapshot
    end
end
```

Pendant que ce thread est actif, nous pouvons commencer à envoyer des commandes dans notre système de ressources humaines :

```
richard = Employee.new('Richard', '1001','1 rue des Rails')
laurent = Employee.new('Laurent','1002','34 avenue Ruby')
store.execute_command(AddEmployee.new(richard))
store.execute_command(AddEmployee.new(laurent))
```

Lorsque Richard et Laurent se retrouvent dans le système de stockage de Madeleine, on peut lancer quelques requêtes :

```
puts(store.execute_command(FindEmployee.new('1001')))
puts(store.execute_command(FindEmployee.new('1002')))
```

Le résultat affiché sera :

```
Employee: name: Richard num: 1001 addr: 1 rue des Rails
Employee: name: Laurent num: 1002 addr: 34 avenue Ruby
```

On peut même changer l'adresse de Richard :

```
store.execute_command(ChangeAddress.new('1001', '55 rue Merb'))
```

Madeleine est un formidable exemple du pattern Command. Au fur et à mesure que les commandes (ajouter un employé ou changer une adresse précise) arrivent dans le système, Madeleine modifie la représentation des données dans la mémoire à l'aide de ce pattern. Mais la commande est également écrite dans un fichier. Lorsque le système s'arrête, Madeleine est capable de restaurer l'état correct en lisant la dernière capture des données et en appliquant toutes les commandes en attente. Au bout d'un intervalle donné – 20 secondes dans notre exemple –, on écrit une nouvelle capture des données courantes et on nettoie toutes les commandes accumulées sur le disque.

En conclusion

Avec le pattern Command nous construisons des objets qui savent accomplir des tâches spécifiques. Le mot clé est ici "spécifique". En effet, une instance de commande du pattern Command ne sait pas changer l'adresse de n'importe quel employé, mais elle est capable de faire déménager un employé précis. Les commandes sont utiles pour maintenir une liste des actions à exécuter ou pour se souvenir des actions déjà effectuées. Vous pouvez également lancer vos commandes dans le sens inverse pour annuler les modifications apportées. Les commandes peuvent être implémentées comme des classes complètes ou de simples blocs de code en fonction de leur complexité.

Le pattern Command a beaucoup de points communs avec le pattern Observer. Les deux identifient un objet – une commande dans le cas de Command et un observateur dans le cas d'Observer – qui est appelé par un autre participant du pattern. Cet objet que je passe au bouton graphique est-il une commande (c'est-à-dire l'action à exécuter lorsque le bouton est sélectionné) ou bien est-ce un observateur qui attend la notification du changement de l'état du bouton ? La réponse est : cela dépend. L'objet commande est capable de faire une action mais ne s'intéresse pas particulièrement à l'état de l'objet qui provoque son exécution. Un observateur est au contraire concerné par l'état du sujet, c'est-à-dire l'objet qui l'a appelé. Command ou Observer : à vous de voir.

9

Combler le fossé avec l'Adapter

Tout comme la plupart des gens qui aiment bricoler des appareils électroniques, je dispose, dans ma cave, d'une boîte remplie à ras bord de pièces électroniques. Sur le haut de la boîte se trouve mon fidèle adaptateur USB-PS2, qui me permet de brancher un clavier USB sur un vieux Pentium 250 cabossé que je conserve encore pour une raison inconnue. Si l'on fouille un peu, on trouve ensuite une couche de convertisseurs de port série vers port parallèle. Et au fond de la boîte on tombe sur des petites pièces d'alimentation électrique noires.

Tous ces gadgets ont un point commun : ils permettent de connecter deux appareils qui aimeraient communiquer mais qui ne peuvent pas le faire pour cause de broches alignées différemment, de taille de prise inadaptée ou encore parce que le voltage de sortie de l'un des appareils serait plus que suffisant pour envoyer l'autre au paradis des gadgets. Bref, ce sont des adaptateurs !

Le monde des logiciels a besoin des adaptateurs davantage que le monde du matériel. Les logiciels ne possèdent pas de caractéristiques physiques : un développeur ne stipule pas que les broches d'un connecteur doivent être espacées d'exactement 1,5 mm. Comme les logiciels sont le fruit de nos idées et puisque nous pouvons coder des interfaces aussi vite que nos doigts tapent au clavier, nous, les développeurs, avons des possibilités quasi illimitées à créer des objets incompatibles, des objets qui aimeraient communiquer, mais qui n'y parviennent pas à cause du désaccord qui règne parmi les interfaces.

Dans ce chapitre nous examinerons des adaptateurs du monde logiciel. Nous verrons comment des adaptateurs nous aident à combler le vide entre des interfaces discordantes. Nous apprendrons également comment utiliser une des fonctionnalités les plus

surprenantes de Ruby – la possibilité de modifier des objets et des classes à la volée, au moment de l'exécution – pour faciliter la tâche de création des adaptateurs.

Adaptateurs logiciels

Pour commencer, imaginons que nous disposons d'une classe permettant de crypter des fichiers :

```
class Encrypter
  def initialize(key)
    @key = key
  end

  def encrypt(reader, writer)
    key_index = 0
    while not reader.eof?
      clear_char = reader.getc
      encrypted_char = clear_char ^ @key[key_index]
      writer.putc(encrypted_char)
      key_index = (key_index +1) % @key.size
    end
  end
end
```

La méthode `encrypt` de la classe `Encrypter` nécessite deux fichiers ouverts, l'un en lecture, l'autre en écriture, ainsi qu'une clé d'encryption. Cette méthode écrit, octet par octet, la version cryptée du fichier d'entrée dans le fichier de sortie.[1]

L'usage de la classe `Encrypter` pour crypter un fichier ordinaire est simple. Il suffit d'ouvrir les deux fichiers et d'appeler `encrypt` avec la clé secrète de votre choix :

```
reader = File.open('message.txt')
writer = File.open('message.encrypted','w')
encrypter = Encrypter.new('my secret key')
encrypter.encrypt(reader, writer)
```

Et maintenant le piège : qu'arrive-t-il si les données à encoder sont contenues dans une chaîne de caractères au lieu d'un fichier ? Dans ce cas, nous aurions besoin d'un objet qui, vu de l'extérieur, prendrait l'allure d'un fichier et qui offrirait donc la même interface qu'un objet Ruby `IO` mais qui, intérieurement, récupérerait ses caractères dans une chaîne. C'est d'un `StringIOAdapter` dont nous avons besoin :

---

1. La classe `Encrypter` applique un vénérable algorithme d'encryption. Pour obtenir le texte crypté elle calcule le OU exclusif (parfois nommé XOR) pour chaque caractère du texte d'entrée avec le caractère correspondant de la clé. La clé est répétée autant de fois que nécessaire. Cet algorithme élégant est complètement autonome : pour décrypter le texte il suffit de réexécuter la procédure sur le texte crypté avec la même clé.

```
class StringIOAdapter
  def initialize(string)
    @string = string
    @position = 0
  end

  def getc
    if @position >= @string.length
      raise EOFError
    end
    ch = @string[@position]
    @position += 1
    return ch
  end

  def eof?
    return @position >= @string.length
  end
end
```

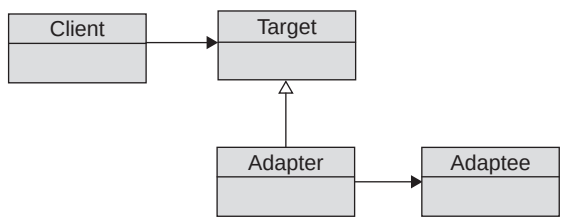
Notre `StringIOAdapter` comprend deux variables d'instance : la référence vers la chaîne de caractères et l'index de la position. À chaque appel de `getc`, `StringIOAdapter` renvoie le caractère à la position courante et incrémente l'index. Lorsqu'il n'y a plus de caractères dans la chaîne, la méthode `getc` lance une exception. La méthode `eof?` retourne `true` s'il n'y a plus de caractères et `false` dans les autres cas.

Pour utiliser `Encrypter` avec `StringIOAdapter` nous remplaçons simplement le fichier d'entrée par l'adaptateur :

```
encrypter = Encrypter.new('XYZZY')
reader = StringIOAdapter.new('We attack at dawn')
writer = File.open('out.txt', 'w')
encrypter.encrypt(reader, writer)
```

Comme vous l'avez deviné, la classe `StringIOAdapter` est un exemple d'adaptateur. Un adaptateur est un objet qui fait le lien entre l'interface existante et l'interface souhaitée.

Figure 9.1
Le pattern Adapter

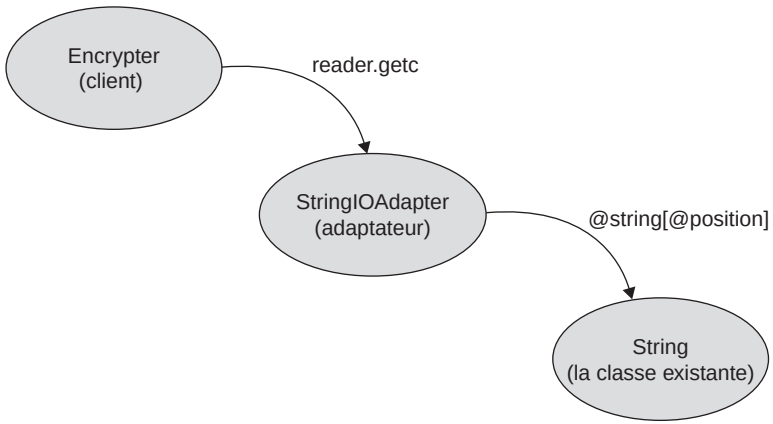


Le diagramme de classes du pattern Adapter est présenté à la Figure 9.1. Voici l'idée principale de ce diagramme : le client a connaissance d'une classe cible – en tant que client j'ai une référence vers mon objet cible. Le client s'attend à trouver une certaine

interface dans la classe cible. Il n'est pas informé qu'en vérité l'objet cible est un adaptateur qui conserve une référence vers un autre objet fournisseur de la fonctionnalité. Dans un monde parfait toutes les interfaces seraient parfaitement compatibles et le client s'adresserait directement à l'objet adapté. Néanmoins, dans le monde réel nous sommes obligés de développer des adaptateurs car l'interface requise par le client ne correspond pas à l'interface proposée par l'objet existant.

Revenons à notre exemple. `Encrypter` est notre objet client, il recherche une référence vers l'objet cible, qui dans ce cas est une instance d'`IO`. En réalité, le client obtient une référence vers l'adaptateur `StringIOAdapter` (voir Figure 9.2).

Figure 9.2
Le fonctionnement de l'objet StringIOAdapter



La classe `StringIOAdapter` ressemble de l'extérieur à un objet `IO` classique, mais elle récupère secrètement ses caractères dans l'objet existant : une chaîne de caractères.

Les interfaces presque parfaites

Les situations probablement les plus frustrantes qui requièrent un adaptateur sont celles où l'interface existante correspond presque – mais pas tout à fait – à l'interface requise. Par exemple, imaginez que nous écrivions une classe pour afficher du texte à l'écran :

```
class Renderer
  def render(text_object)
    text = text_object.text
    size = text_object.size_inches
    color = text_object.color
    # afficher le texte ...
  end
end
```

Il est clair que l'objet `Renderer` s'attend à afficher un objet qui ressemble à ceci :

```
class TextObject
  attr_reader :text, :size_inches, :color
  def initialize(text, size_inches, color)
    @text = text
    @size_inches = size_inches
    @color = color
  end
end
```

Malheureusement, on découvre qu'une partie du texte à afficher est encapsulée dans un objet qui ressemble plutôt à ceci :

```
class BritishTextObject
  attr_reader :string, :size_mm, :colour
    # ...
end
```

La bonne nouvelle, c'est que `BritishTextObject` contient tout ce qui est fondamentalement nécessaire pour afficher du texte. La mauvaise, c'est que le champ qui encapsule le texte s'appelle `string` au lieu de `text`, la taille du texte est en millimètre plutôt qu'en pouce et, qui plus est, l'attribut `colour` s'écrit chez nos amis grands-bretons avec la lettre "u".

Pour résoudre le problème, on peut certainement recourir au pattern Adapter :

```
class BritishTextObjectAdapter < TextObject
  def initialize(bto)
    @bto = bto
  end

  def text
    return @bto.string
  end

  def size_inches
    return @bto. size_itim / 25.4
  end

  def color
    return @bto.colour
  end
end
```

C'est une possibilité... Mais on pourrait tirer parti de la capacité qu'a Ruby à modifier des classes à la volée.

Une alternative adaptative ?

Si vous êtes débutant en Ruby, vous pensez probablement que c'est un langage plutôt conventionnel avec son héritage simple ainsi que ses classes, ses méthodes et ses opérateurs `if` incorporés. Certes, les blocs de code semblent légèrement étranges mais, finalement, ils se transforment en objets `Proc` qui se comportent de façon assez familière. Si c'est ce que vous pensez, accrochez-vous avant de lire ce qui suit : en Ruby, toute classe est modifiable à tout moment !

Pour comprendre les implications de ce principe, imaginez que l'on décide de ne pas faire le lien entre l'instance de `BritishTextObject` existante et l'interface requise de `TextObject`. Nous changerons plutôt l'objet `BritishTextObject` afin qu'il possède l'interface nécessaire. Pour y parvenir, il faut tout d'abord s'assurer que la classe `BritishTextObject` est chargée et, ensuite, nous ouvrons la classe et lui ajoutons quelques méthodes :

```
# S'assurer que la classe initiale est chargée
require 'british_text_object'

# Rajouter des méthodes à la classe initiale
class BritishTextObject
  def color
    return colour
  end

  def text
    return string
  end

  def size_inches
    return size_mm / 2 5.4
  end
end
```

Le fonctionnement de ce code est très simple. La méthode `require` en début de fichier charge la classe initiale `BritishTextObject`. L'instruction `class BritishTextObject` après l'appel `require` ne crée pas une nouvelle classe, mais ouvre bel et bien la classe existante pour lui ajouter quelques méthodes. Vous êtes toujours là ? Les modifications des classes ne sont limitées en aucune manière. Il est non seulement possible d'ajouter des méthodes, mais on peut aussi modifier et même complètement supprimer des méthodes existantes. Le plus surprenant vient peut-être du fait que toutes ces opérations sont possibles aussi bien sur les classes standard de Ruby que sur vos propres classes. On pourrait, par exemple, vandaliser la méthode `abs` de `Fixnum` :

```
# Ne faites jamais ceci !
class Fixnum
```

```
  def abs
    return 42
  end
end
```

Selon la version "améliorée" d'`abs`, la valeur absolue de n'importe quel `Fixnum` est égale à 42.

Donc, les deux instructions

```
puts(79.abs)
puts(-1234.abs)
```

affichent

```
42
42
```

La possibilité de modifier des classes est un des secrets qui procurent à Ruby sa flexibilité et sa puissance. Toutefois, comme l'illustre l'exemple précédent, une grande puissance implique de grandes responsabilités.

Modifier une instance unique

Puisque modifier une classe entière à la volée paraît une solution quelque peu extrême, Ruby propose une alternative moins invasive. Au lieu de modifier une classe, on peut apporter des changements au comportement d'une instance donnée :

```
bto = BritishTextObject.new('hello', 50.8, :blue)

class << bto
  def color
    colour
  end

  def text
    string
  end

  def size_inches
    return size_mm/25.4
  end
end
```

La ligne clé est ici :

```
class << bto
```

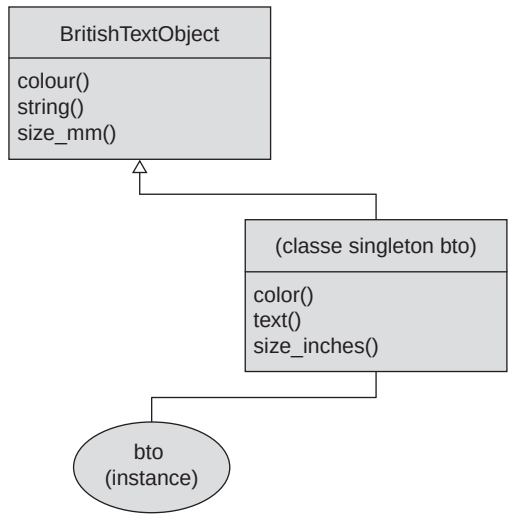
Ce code est essentiellement une instruction permettant de modifier le comportement de l'objet `bto` indépendamment de sa classe. Une autre syntaxe est disponible : pour obtenir le même effet, on peut simplement définir des méthodes sur une instance :

```
def bto.color
  colour
end

def bto.text

  string
end
# ...
```

Ruby nomme les méthodes propres à un objet donné "des méthodes singleton"[1]. Il s'avère qu'une majorité des objets Ruby[2] possèdent une classe plus ou moins secrète en plus de leur classe normale. Comme le montre la Figure 9.3, cette seconde classe *singleton* est en fait le premier endroit que l'interpréteur Ruby inspecte lorsqu'une méthode est appelée. Toute méthode définie dans une classe singleton surcharge les méthodes de la classe habituelle[3]. Le code précédent modifie la classe singleton de l'objet `bto`. Les changements sont apportés avec une discrétion extrême car même après modification l'objet se considère toujours comme une instance de la classe d'origine[4].

Figure 9.3

Les méthodes Singleton de l'instance de BritishTextObject



1. Le nom *singleton* est un terme regrettable. Ces méthodes n'ont rien en commun avec le pattern Singleton, que nous examinerons au Chapitre 12.
2. Les objets immuables – par exemple des instances de `Fixnum` – n'acceptent pas l'ajout des méthodes singleton.
3. Les méthodes singleton surchargent également les méthodes de tout module inclus dans la classe.
4. Pas si discret que cela, finalement : on peut se renseigner sur des méthodes singleton d'un objet à l'aide de la méthode `singleton_methods`.

Adapter ou modifier ?

Il est indéniable que le code gagne en simplicité lorsque le développeur opte pour la modification de la classe ou de son instance plutôt que pour la création d'un adaptateur. Si l'on modifie l'objet d'origine ou sa classe, il n'y a nul besoin de créer une classe adaptateur supplémentaire ni de se soucier de la manière d'encapsuler l'objet existant dans l'adaptateur. Les choses fonctionnent. Pourtant, cette technique de modification implique des atteintes sérieuses à l'encapsulation : vous plongez à l'intérieur d'une classe ou d'un objet et vous en modifiez l'implémentation. Dans quelle situation utiliser un adaptateur et quand est-il acceptable de remanier la tuyauterie interne d'une classe ?

Comme toujours, un peu de pragmatisme reste la meilleure recette. Préférez la modification d'une classe dans les circonstances suivantes :

- Les modifications sont simples et claires. Les alias de méthodes dans le code d'exemple donné ci-dessus en sont un parfait exemple.
- Vous comprenez bien la classe à modifier ainsi que son usage. Une intervention chirurgicale sur une classe que vous n'avez pas étudiée au préalable mènerait probablement à des problèmes.

Préférez la solution adaptateur dans les situations suivantes :

- Les discordances entre interfaces sont complexes et étendues. Par exemple, vous ne devriez probablement pas essayer de modifier une chaîne de caractères afin de lui affecter l'interface de `Fixnum`.
- Vous ne comprenez pas le fonctionnement de la classe. L'ignorance est une sérieuse raison pour rester prudent.

L'ingénierie est la science des compromis. Les adaptateurs préservent l'encapsulation au prix d'une certaine complexité. Vous gagnez en simplicité si vous modifiez la classe, mais vous êtes obligé de plonger dans les détails de son implémentation.

User et abuser du pattern Adapter

Un des avantages du typage à la canard de Ruby réside dans la possibilité de créer des adaptateurs pour une partie de l'interface cible effectivement utilisée par le client. Par exemple, les objets `IO` proposent un grand nombre de méthodes. Un vrai objet `IO` vous permet de lire des lignes, de faire des recherches dans un fichier ainsi qu'un tas d'autres

choses liées aux fichiers. Mais notre objet `StringIOAdapter` implémente exactement deux méthodes : `getc` et `eof?`. C'est suffisant dans notre exemple car ce sont les deux méthodes de la classe `IO` réellement utilisées par la classe `Encrypter`. Les adaptateurs avec des implémentations partielles sont des lames à double tranchant : d'une part, il est pratique d'implémenter le strict minimum mais, d'autre part, votre programme ne fonctionnerait plus si le client décidait soudainement d'appeler une méthode que vous n'avez pas jugé utile d'implémenter.

Le pattern Adapter dans le monde réel

On peut trouver un exemple classique du pattern Adapter dans ActiveRecord, le système de mappage objet relationnel de Ruby on Rails. ActiveRecord doit gérer tout un éventail de systèmes de bases de données différentes : MySQL, Oracle et Postgres, sans parler de SQLServer. Tous ces systèmes fournissent une API Ruby, ce qui est bien. Mais toutes les API sont différentes, ce qui est très ennuyeux. Par exemple, lorsque vous établissez une connexion vers une base de données MySQL et que vous devez exécuter du code SQL, il faut appeler la méthode `query` :

```
results = mysql_connection.query(sql)
```

Mais, si vous utilisez Sybase, la méthode à appeler est `sql` :

```
results = sybase_connection.sql(sql)
```

Et, si vous avez affaire à Oracle, il faut appeler la méthode `execute` pour obtenir en retour un pointeur vers le résultat et non le résultat lui-même. On dirait que les éditeurs de bases de données ont comploté pour s'assurer que les systèmes ne soient pas compatibles.

ActiveRecord gère ces différences par une interface standardisée, encapsulée dans la classe `AbstractAdapter`. Cette classe définit une interface vers la base de données. Cette interface unique est utilisée partout dans ActiveRecord. Par exemple, `AbstractAdapter` expose une méthode standard `select_all` pour exécuter une requête SQL `select` et retourner le résultat. Une sous-classe de la classe `AbstractAdapter` est disponible pour chacune des bases de données : il existe `MysqlAdapter`, `OracleAdapter` ainsi que `SybaseAdapter`. Chaque adaptateur implémente la méthode `select_all` en se fondant sur l'API du système de base de données correspondant.

Enfin, notre exemple `StringIOAdapter` est inspiré par la classe `StringIO`, qui est distribuée avec Ruby.

En conclusion

Un adaptateur n'a rien de magique. C'est un objet qui absorbe la différence qui peut exister entre des interfaces requises et des objets existants. Un adaptateur expose au monde extérieur l'interface nécessaire, mais cette interface est implémentée en faisant des appels à un objet caché à l'intérieur. Cet objet fournit toute la fonctionnalité nécessaire, mais à l'aide d'une interface inadaptée.

Ruby propose également une deuxième manière – plus limitée – de résoudre le problème d'interface inadaptée : nous pouvons simplement modifier l'objet au moment de l'exécution pour lui attribuer l'interface requise. Formulé autrement, nous pouvons forcer l'objet à se soumettre. Le choix entre adaptateur ou modification dynamique d'un objet se détermine par votre compréhension de la classe concernée et les considérations liées à l'encapsulation. Si vous maîtrisez le fonctionnement de la classe et que les changements d'interface soient relativement mineurs, modifier l'objet peut être la bonne solution. Si l'objet est complexe et si vous ne comprenez pas complètement son fonctionnement, optez pour un adaptateur classique.

Le pattern Adapter est le premier membre d'une famille que nous allons étudier dans les chapitres qui suivent : la famille des patterns dans lesquels un objet en remplace un autre. Cette famille d'imposteurs orientés objet inclut également des proxies (mandataires) et des décorateurs. Dans les deux cas un objet agit en tant que porte-parole d'un autre objet. Comme vous le verrez par la suite, le code de ces patterns se ressemble. Au risque de me répéter, n'oubliez pas qu'un pattern ne se réduit pas à un bout de code : l'intention est cruciale. Un adaptateur est seulement un adaptateur dans le cas où vous vous retrouvez avec des objets dont les interfaces sont inadaptées et que vous voulez isoler l'effort de gestion de ces interfaces incompatibles dans votre système.

10

Créer un intermédiaire pour votre objet avec Proxy

Le monde du génie logiciel ne manque pas d'ironie. On peut travailler jour et nuit pendant des mois pour créer un objet `BankAccount` (un pur chef-d'œuvre technique qui permet aux clients de gérer leurs comptes bancaires) et passer quelques mois de plus pour restreindre l'accès à cet objet sauf pour quelques utilisateurs autorisés, pour finalement s'entendre dire par le patron que les utilisateurs autorisés n'ont même pas besoin de l'objet `BankAccount`. Ou, tout au moins, ils n'en ont pas besoin sur leurs machines : ce serait bien s'il pouvait utiliser la classe `BankAccount` à distance à partir d'un serveur car les utilisateurs ne voudront jamais installer cet objet sur leur ordinateur.

Et, bien entendu, c'est au moment où on commence à entendre dans la bouche du patron l'argument massue *"il nous faut ce code tout de suite"* qu'une toute nouvelle demande nous arrive : au moment de l'exécution il faut retarder autant que possible la création des objets `BankAccount` pour des raisons de performance.

Aussi indépendants que ces problèmes puissent paraître – contrôler l'accès à un objet, fournir un moyen d'accès à l'objet quel que soit son emplacement ou retarder sa création –, les trois ont une solution commune : le pattern Proxy.

Dans ce chapitre, nous allons étudier le pattern Proxy, examiner la façon traditionnelle de l'implémenter et essayer de l'appliquer pour résoudre nos trois problèmes. Enfin, nous allons sortir de notre chapeau magique une technique Ruby qui rend la tâche de développement d'un proxy aussi facile à écrire qu'une simple méthode.

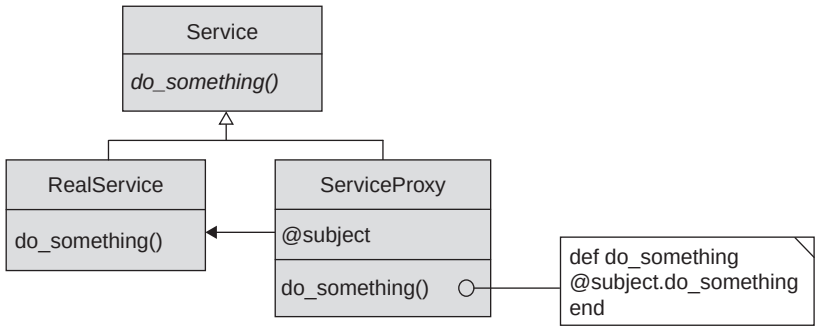


Figure 10.1
Le pattern Proxy

Les proxies à la rescousse

Le pattern Proxy se fonde essentiellement sur un petit mensonge. Lorsqu'un client sollicite un objet – par exemple l'objet `BankAccount` –, il faut effectivement lui retourner un objet. Néanmoins, l'objet retourné n'est pas tout à fait l'objet que le client s'attend à recevoir. Nous retournons un objet imposteur qui fournit une interface identique à celle de l'objet attendu. Les membres du GoF ont nommé cet objet contrefait un *proxy* (voir Figure 10.1). Il contient une référence vers le véritable objet, le *sujet*, qui est caché à l'intérieur. Lorsque le code client appelle une méthode sur le proxy, l'appel est transféré vers le vrai objet.

Pour concrétiser cette idée, faisons un peu de finance. Le code ci-après est une classe qui sert à garder un suivi d'un compte bancaire :

```
class BankAccount
  attr_reader :balance
  def initialize(starting_balance=0)
    @balance = starting_balance
  end

  def deposit(amount)
    @balance += amount
  end

  def withdraw(amount)
    @balance -= amount
  end
end
```

Les instances de `BankAccount` seront nos vrais objets, ce que nous appelons nos sujets. Définissons un proxy pour `BankAccount` :

```
class BankAccountProxy
  def initialize(real_object)
    @real_object = real_object
  end
```

```
  def balance
    @real_object.balance
  end

  def deposit(amount)
    @real_object.deposit(amount)
  end

  def withdraw(amount)
    @real_object.withdraw(amount)
  end
end
```

Maintenant, on peut créer un objet `BankAccount` ainsi qu'un proxy et les utiliser comme objets interchangeables :

```
account = BankAccount.new(100)
account.deposit(50)
account.withdraw(10)
proxy = BankAccountProxy.new(account)
proxy.deposit(50)
proxy.withdraw(10)
```

Rien d'extraordinaire ne se passe dans `BankAccountProxy`. Cette classe présente exactement la même interface que son sujet, l'objet `BankAccount`. Mais le proxy ne maîtrise pas la finance. Lorsqu'une de ses méthodes est appelée, l'objet `BankAccountProxy` délègue l'appel de méthode à son sujet `BankAccount`.

Évidemment, si notre proxy ne faisait qu'envoyer bêtement un écho des appels de méthodes vers le sujet, le seul résultat que nous aurions accompli, c'est d'imposer une charge supplémentaire au processeur. Mais avec un proxy nous avons un objet intermédiaire entre le client et le vrai objet. Si l'on voulait contrôler l'accès à un compte bancaire, le proxy serait l'endroit parfait pour ce type de code.

Un proxy de contrôle d'accès

Transformons notre `BankAccountProxy` générique qui ne fait rien en un *proxy de contrôle d'accès* au sujet. Pour y arriver, il suffit d'ajouter une vérification au début de chaque méthode :

```
require 'etc'
class AccountProtectionProxy
  def initialize(real_account, owner_name)
    @subject = real_account
    @owner_name = owner_name
  end
```

```
    def deposit(amount)
      check_access
      return @subject.deposit(amount)
    end

    def withdraw(amount)
      check_access
      return @subject.withdraw(amount)
    end

    def balance
      check_access
      return @subject.balance
    end

    def check_access
      if Etc.getlogin != @owner_name
        raise "Illegal access: #{Etc.getlogin} cannot access account."
      end
    end
  end
```

Chaque opération sur le compte est protégée par un appel à la méthode `check_access`. La méthode `check_access` porte bien son nom : elle vérifie que l'utilisateur courant est autorisé à accéder au compte. La version de `check_access` utilisée dans notre exemple se sert du module `Etc`[1] pour récupérer le nom de l'utilisateur courant. Ce nom est ensuite comparé avec le nom du propriétaire du compte qui est passé dans le constructeur.

Évidemment, on aurait pu inclure le code de vérification directement dans l'objet `BankAccount`. Mais l'utilisation d'un proxy de contrôle d'accès nous donne un avantage : une bonne séparation de responsabilités. Le proxy décide qui a le droit d'effectuer l'action demandée. La seule chose dont un objet `BankAccount` doit s'occuper, c'est de gérer le compte bancaire. L'implémentation du contrôle d'accès dans un proxy nous permet de remplacer facilement l'algorithme de gestion de sécurité (il suffit d'encapsuler le sujet dans un proxy différent) ou de supprimer ce niveau de sécurité complètement (ne pas utiliser de proxy). Inversement, nous pouvons modifier l'implémentation de l'objet `BankAccount` sans affecter la gestion de sécurité.

Les proxies de contrôle d'accès présentent aussi l'avantage de séparer proprement la fonction de protection de la logique métier de l'objet réel, minimisant ainsi le risque qu'une information importante transpire à travers la couche de protection.

---

1. Le module `Etc` est plus ou moins standard dans Ruby. Il fait partie de la distribution Ruby pour des systèmes UNIX et des systèmes semblables. La version Windows est facultative, mais elle est largement disponible. La version Windows s'installe en un clic à l'aide de l'installeur Ruby, je vais donc partir du principe que vous l'avez.

Des proxies distants

Pour certaines applications il se peut que la sécurité ne soit pas votre souci premier mais que le nœud de l'affaire soit relatif à l'emplacement où l'application va s'exécuter. Imaginons que vous ayez un programme sur une machine client et que vous vouliez utiliser l'objet `BankAccount` mais que cet objet se trouve sur un serveur très loin sur le réseau. L'une des solutions consisterait à faire travailler le programme client : il faudrait alors créer et envoyer des paquets et donc gérer toute la complexité de la communication à travers le réseau (potentiellement instable). L'alternative consiste à cacher la complexité derrière un proxy, c'est-à-dire un objet qui réside sur une machine client et présente au code client exactement la même interface qu'un objet `BankAccount`. Lorsqu'une requête arrive, le proxy prend en charge tout le travail d'empaquetage de la requête, l'envoie sur le réseau, attend la réponse, la décode et la retourne au client, qui n'y voit que du feu.

Du point de vue du client, il fait appel à ce qu'il pense être un vrai objet `BankAccount`, et plus tard (probablement beaucoup plus tard) il obtient la réponse. Quasiment tous les systèmes d'appels de procédures distants (RPC ou *Remote Procedure Call* en anglais) fonctionnent de cette façon.

Voici un court exemple d'un proxy distant. Le code suivant utilise le client SOAP livré avec Ruby pour créer un proxy distant d'un service SOAP public qui présente des informations météorologiques[1] :

```
require 'soap/wsdlDriver'
wsdl_url = 'http://www.webservicex.net/WeatherForecast.asmx?WSDL'
proxy = SOAP::WSDLDriverFactory.new( wsdl_url ).create_rpc_driver
weather_info = proxy.GetWeatherByZipCode('ZipCode'=>'19128')
```

Une fois le proxy configuré, le client ne se préoccupe plus du fait que le service réside réellement à l'adresse `www.webservicex.net`. Il appelle simplement `GetWeatherByZipCode` et laisse le proxy gérer les détails de la communication réseau.

Des proxies distants offrent en partie les mêmes avantages que des proxies de contrôle d'accès. Plus particulièrement, un proxy distant permet de séparer les responsabilités : le sujet peut se concentrer sur les prévisions météorologiques ou toute autre tâche métier et laisser le soin à un autre objet – le proxy – de se charger de la logistique de transfert des octets sur le réseau. Changer de protocole de communication (par exemple

1. Si vous décidez de tester cet exemple, souvenez-vous que les services Web publics ont une espérance de vie extrêmement courte, donc le service utilisé dans l'exemple peut ne pas être accessible lorsque vous essaierez d'y accéder.

passer de SOAP à XMLRPC) devient alors un jeu d'enfant puisqu'il suffit de remplacer le proxy.

Des proxies virtuels à vous rendre paresseux

Nous pouvons aussi recourir aux proxies pour retarder la création coûteuse de certains objets jusqu'au moment où ils deviennent absolument nécessaires. C'est précisément la troisième demande que l'on devait gérer dans l'exemple qui ouvre ce chapitre. Souvenez-vous que la dernière exigence dans notre projet financier était de repousser la création des instances de `BankAccount` aussi longtemps que possible. Il ne faut pas créer un vrai compte bancaire tant que l'utilisateur n'est pas prêt à l'utiliser : par exemple pour déposer de l'argent. Mais contaminer le code client avec toute la complexité liée à ce retard de création n'est pas une bonne idée non plus. La solution vient cette fois d'un autre type de proxy : un *proxy virtuel*.

Le proxy virtuel est en quelque sorte le plus grand imposteur de tous les proxies. Il se fait passer pour un véritable objet alors qu'il ne le connaît même pas et qu'il n'en aura aucune connaissance. Ce n'est qu'au moment où certaines méthodes seront sollicitées que le proxy virtuel se précipitera pour créer le véritable objet (où récupérer l'accès au vrai objet par un autre moyen).

Implémenter un proxy virtuel est une chose très simple :

```
class VirtualAccountProxy
  def initialize(starting_balance=0)
    @starting_balance = starting_balance
  end

  def deposit(amount)
    s = subject
    return s.deposit(amount)
  end

  def withdraw(amount)
    s = subject
    return s.withdraw(amount)
  end

  def balance
    s = subject
    return s.balance
  end

  def subject
    @subject ||(@subject = BankAccount.new(@starting_balance))
  end
end
```

Le cœur de notre `VirtualAccountProxy` est la méthode `subject`. Cette méthode vérifie si l'objet `BankAccount` a déjà été créé et le crée le cas échéant. Pour y parvenir, la méthode `subject` utilise un idiome très courant en Ruby mais à la syntaxe légèrement étrange :

```
@subject || (@subject = BankAccount.new(@starting_balance))
```

Cette ligne représente essentiellement une grande expression OR. Le premier élément de l'expression OR est `@subject`. Si la valeur de `@subject` n'est pas `nil`, l'expression s'évalue à cette valeur (dans notre cas, c'est l'objet `BankAccount`). Si `@subject` est à `nil`, Ruby évalue la partie droite de l'expression OR qui crée un nouvel objet `BankAccount`. Ce nouvel objet devient donc le résultat de l'expression.

Dans cette implémentation, `VirtualAccountProxy` est responsable de l'instanciation d'un nouvel objet `BankAccount`, ce qui est incontestablement un défaut. Cette approche augmente sérieusement le niveau de couplage entre le sujet et le proxy. La stratégie peut être améliorée en appliquant la magie des blocs Ruby :

```
class VirtualAccountProxy
  def initialize(&creation_block)
    @creation_block = creation_block
  end
  # Les autres méthodes sont omises...
  def subject
    @subject || (@subject = @creation_block.call)
  end
end
```

Dans la nouvelle implémentation, le code de création de proxy passe un bloc qui a pour mission de créer le compte bancaire au bon moment :

```
account = VirtualAccountProxy.new { BankAccount.new(10) }
```

Tout comme les deux autres types de proxies, les proxies virtuels permettent d'atteindre la séparation de responsabilités : le vrai objet `BankAccount` gère les transferts d'argent, alors que `VirtualAccountProxy` s'occupe de la création de l'objet `BankAccount`.

Éliminer les méthodes ennuyeuses des proxies

Tous les proxies vus jusqu'ici partagent une même caractéristique quelque peu embêtante : la nécessité d'écrire toutes les méthodes proxy de façon répétitive. Par exemple, tous les proxies d'un compte bancaire sont obligés d'implémenter les méthodes `deposit`, `withdraw` et `balance`. Bien évidemment, le nombre de méthodes peut être bien plus grand que cela. La classe `Array` de Ruby, par exemple, compte 118 méthodes et

`String`, 142. Écrire 142 méthodes proxy n'est pas seulement une corvée, c'est aussi extrêmement propice aux erreurs.

Est-il possible d'éviter d'écrire toutes ces méthodes fort ennuyeuses ? Il s'avère que Ruby propose une solution. Cette solution est ancrée dans les notions que vous avez apprises très tôt dans votre carrière orientée objet et que vous avez probablement oubliées depuis longtemps.

Les méthodes et le transfert des messages

Si votre introduction à la programmation orientée objet ressemblait à la mienne, la première journée de cours a dû vous apprendre la notion de *transfert des messages*. On vous a probablement dit que

```
account.deposit(50)
```

signifie que vous envoyez un message `deposit` à l'objet `account`. Évidemment, si vous avez utilisé par la suite un langage statiquement typé, vous avez rapidement compris que `account.deposit (50)` était en fait un synonyme d'appel de méthode `deposit` sur l'objet `account`. Le système entier du typage statique était là pour assurer que la méthode `deposit` était bien présente et qu'elle était bien appelée. Donc, à la fin de la première journée de notre apprentissage de la programmation orientée objet, nous nous sommes tous arrêtés de parler de *transfert de messages* et avons commencé à raisonner en termes d'appels de méthodes.

Le concept du transfert de messages aurait un sens si, en appelant `account.deposit(50)`, la classe `BankAccount` était libre de faire une action autre que simplement appeler la méthode `deposit`. Par exemple, la classe `BankAccount` devrait pouvoir utiliser une autre méthode plutôt que `deposit` ou bien, au contraire, décider de ne rien faire du tout. Il s'avère que Ruby rend toutes ces options possibles.

La signification réelle de `account.deposit(50)` en Ruby est plus proche du transfert de message que le modèle d'appel de méthode direct utilisé dans la majorité des langages statiquement typés. Lorsqu'on invoque `account.deposit(50)`, Ruby commence le traitement de manière classique : il recherche la méthode `deposit` dans la classe `BankAccount`, puis dans sa classe mère, etc. jusqu'à trouver la méthode ou jusqu'à atteindre le sommet de l'arbre hiérarchique des classes. Si la méthode est trouvée, on obtient le comportement habituel : la méthode `deposit` est appelée et le compte est crédité de 50 euros.

Mais qu'arrive-t-il s'il n'y a pas de méthode `deposit` ? Dans ce cas, Ruby suit un scénario un peu plus inhabituel : il appelle une autre méthode. Cette méthode du dernier

recours est nommée `method_missing`. Une fois de plus, Ruby part à la recherche de cette méthode dans la classe `BankAccount`. Si cette méthode n'existe pas dans `BankAccount`, Ruby remonte l'arbre d'héritage des classes vers sa classe mère jusqu'à trouver la méthode ou bien jusqu'à atteindre la classe `Object`. La recherche se termine dans `Objet`, car cette classe est équipée de la méthode `method_missing`. Son implémentation lance simplement l'exception `NoMethodError`.

Cette approche implique que, si vous ne définissez pas la méthode `method_missing` dans votre classe (ou dans une classe parent), le code fonctionne de manière classique : on appelle une mauvaise méthode et Ruby lance une exception. La beauté de la méthode `method_missing` tient au fait que lorsque vous l'implémentez dans votre classe cette classe est capable de traiter n'importe quel appel de méthode – ou plutôt envoi de message – et d'effectuer une action adaptée à la situation[1]. C'est un réel transfert de messages.

La méthode method_missing

La méthode `method_missing` fait partie des méthodes à liste d'arguments variable que nous avons brièvement vues au Chapitre 2. Le premier argument est toujours un symbole : le nom de la méthode inexistante. Il est suivi par les arguments de l'appel de méthode initial. Jetons un œil sur un simple exemple, une classe qui lance sa propre exception lorsqu'une méthode inexistante est appelée. Essayez d'exécuter le code suivant :

```
class TestMethodMissing
  def hello
    puts("Hello from a real method")
  end

  def method_missing(name, *args)
    puts("Warning, warning, unknown method called: #{name}")
    puts("Arguments: #{args.join(' ')}")
  end
end
```

Si l'on envoie à l'instance de `TestMethodMissing` un message correspondant à une vraie méthode,

```
tmm = TestMethodMissing.new
tmm.hello
```

1. Le langage de programmation Smalltalk se comporte quasiment de la même façon que Ruby lorsqu'un client appelle une méthode non existante. Mais le nom de la méthode du dernier recours en Smalltalk est beaucoup plus parlant : `doesNotUnderstand`.

son comportement est habituel :

```
Hello from a real method
```

Mais, lorsque l'on appelle une méthode inexistante, l'appel est transféré dans `method_missing` :

```
tmm.goodbye('cruel', 'world')
Warning, warning, unknown method called: goodbye
Arguments: cruel world
```

Envoi des messages

L'idée de l'envoi de messages est incorporée très profondément dans Ruby. On peut non seulement récupérer des messages inattendus à l'aide de `method_missing`, mais aussi envoyer des messages à des objets explicitement avec la méthode `send`. Par exemple, l'envoi des messages

```
tmm.send(:hello)
tmm.send(:goodbye, 'cruel', 'world')
```

provoque le même résultat que les appels normaux des méthodes `hello` et `goodbye` :

```
Hello from a real method
Warning, warning, unknown method called: goodbye
Arguments: cruel world
```

Les arguments à passer sont identiques à ceux de la méthode `method_missing`. Le premier argument est le nom du message. Il est suivi des autres paramètres.

Vous vous demandez peut-être à quoi est due toute cette excitation ? D'accord, on peut gérer des appels de méthodes inexistantes avec `method_missing`. D'accord, on peut passer des messages explicitement. Mais pourquoi choisir d'utiliser l'appel `account.send(:deposit, 50)` alors qu'`account.deposit(50)` est plus court et plus familier ? Eh bien, il se trouve que l'appel par transfert de messages rend l'implémentation des proxies ainsi que des nombreux autres patterns bien plus simple.

Proxies sans peine

Souvenez-vous qu'avant de partir dans la discussion sur le transfert des messages nous déplorions le fait que l'implémentation des proxies impliquait une répétition pénible de toutes les méthodes offertes par les sujets. Et si l'on n'avait plus besoin de les répéter ? Si l'on développait la classe proxy sans définir toutes les méthodes du sujet ? Essayons :

```
class AccountProxy
  def initialize(real_account)
```

```
      @subject = real_account
    end
  end
```

Cette implémentation de notre `AccountProxy` est assez inutile. Si l'on crée cette classe et qu'on lui lance des appels des méthodes définies dans `BankAccount`,

```
ap = AccountProxy.new( BankAccount.new(100) )
ap.deposit(25)
```

on n'aura qu'un grand regard vide au retour :

```
proxy1.rb:32: undefined method 'deposit'
  for #<AccountProxy:0x401bd408>(NoMethodError)
```

Grâce à notre discussion dans la section précédente, on connaît désormais le mécanisme du fonctionnement de ce code. Tout d'abord, Ruby recherche la méthode `deposit` et faute de la trouver il poursuit sa recherche par la méthode `method_missing`, jusqu'à la trouver dans la classe `Object`. Cette méthode de la classe `Object` déclenche l'exception `NoMethodError`.

Et voici l'idée maîtresse : si l'on ajoute la méthode `method_missing` à notre classe proxy, elle sera capable de traiter tout appel de méthode. Le proxy peut également transférer les messages qu'il est incapable de gérer vers de vrais objets `BankAccount` à l'aide de la méthode `send`. Voici la version améliorée de la classe `AccountProxy` :

```
class AccountProxy
  def initialize(real_account)
    @subject = real_account
  end

  def method_missing(name, *args)
    puts("Delegating #{name} message to subject.")
    @subject.send(name, *args)
  end
end
```

Nous sommes maintenant dans la position de passer n'importe quel message de l'objet `BankAccount` au proxy en toute sérénité. Tous les messages que `AccountProxy` ne comprend pas seront transférés à la méthode `method_missing`, qui à son tour les transmettra à l'objet `BankAccount`. On peut maintenant créer et commencer à utiliser le nouveau proxy :

```
ap = AccountProxy.new( BankAccount.new(100) )
ap.deposit(25)
ap.withdraw(50)
puts("account balance is now: #{ap.balance}")
```

Avec le résultat suivant :

```
delegating deposit method to subject,
delegating withdraw method to subject,
delegating balance method to subject,
account balance is now: 75
```

Cette technique offre une méthode de délégation sans peine : exactement ce dont nous avons besoin pour faciliter le processus de développement des proxies. On peut réécrire `AccountProtectionProxy` en utilisant l'approche `method_missing` :

```
class AccountProtectionProxy
  def initialize(real_account, owner_name)
    @subject = real_account
    @owner_name = owner_name
  end

  def method_missing(name, *args)
    check_access
    @subject.send(name, *args )
  end

  def check_access
    if Etc.getlogin != @owner_name
      raise "Illegal access: #{Etc.getlogin} cannot access account."
    end
  end
end
```

Il faut remarquer deux points intéressants dans la nouvelle version d'`AccountProtectionProxy`, le premier est évident mais le deuxième est un peu plus subtil. Tout d'abord, il ne vous a pas échappé qu'`AccountProtectionProxy` n'occupe plus que quinze lignes et que la classe gardera cette taille quel que soit le nombre de méthodes de l'objet réel. Il est moins évident qu'`AccountProtectionProxy` ne comprenne plus aucun code spécifique à `BankAccount`. `AccountProtectionProxy` fonctionnera (et appliquerait la même politique de sécurité) sur tout objet que vous décidez de lui passer.

Pour vous donner un exemple simple, imaginez que nous voulions utiliser `AccountProtectionProxy` pour sécuriser une chaîne de caractères. Si l'on encapsule la chaîne dans notre proxy avec un utilisateur correct (moi !), le code fonctionne correctement :

```
s = AccountProtectionProxy.new( "a simple string", 'russ' )
puts("The length of the string is #{s.length}")
```

Mais si le propriétaire de la chaîne est Fred,

```
s = AccountProtectionProxy.new( "a simple string", 'fred' )
puts("The length of the string is #{s.length}")
```

on ne peut pas y accéder :

```
string_permission.rb:17.in `check_access':
  Illegal access: russ cannot access account.
```

On peut tout aussi simplement définir un proxy virtuel en utilisant `method_missing` :

```
class VirtualProxy
  def initialize(&creation_block)
    @creation_block = creation_block
  end

  def method_missing(name, *args)
    s = subject
    s.send(name, *args)
  end

  def subject
    @subject = @creation_block.call unless @subject
    @subject
  end
end
```

Tout comme notre proxy de contrôle d'accès fondé sur `method_missing`, le deuxième proxy virtuel est universel. On peut l'utiliser par exemple pour retarder la création d'un tableau :

```
array = VirtualProxy.new { Array.new }
array << 'hello'
array << 'out'
array << 'there'
```

Comme vous le verrez par la suite, la méthode `method_missing` peut être pratique dans de nombreuses situations qui font appel à la délégation.

User et abuser du pattern Proxy

Lorsqu'on construit un proxy, particulièrement avec la méthode `method_missing`, il est facile de tomber dans le piège qui consiste à oublier que tout objet est créé avec un nombre minimal de méthodes, celles héritées de la classe `Object`. C'est ainsi que tout objet hérite de la classe `Object` la méthode `to_s`. L'appel vers `to_s` sur la majorité des objets retourne une chaîne de caractères qui contient une description de cet objet. La mission d'un proxy est de se faire passer pour son sujet mais, si l'on appelle `to_s` sur un de nos proxies, l'illusion disparaît aussitôt :

```
account = VirtualProxy.new { BankAccount.new }
puts(account)
#<VirtualProxy:0x40293b48>
```

Le problème, c'est que nous appelons la méthode `to_s` de la classe `VirtualProxy` et non pas celle de `BankAccount`. C'est peut-être le comportement souhaité mais lorsque vous développez des proxies il est important de se rappeler les méthodes d'`Object` si souvent oubliées.

La technique `method_missing` glorifiée dans ce chapitre a également quelques inconvénients. Par exemple, l'utilisation de `method_missing` affecte le niveau de performance. Si l'on compare une classe avec une méthode directe

```
class DirectCall
  def add(a, b)
    a+b
  end
end
```

et une classe qui utilise la méthode `method_missing`

```
class MethodMissingCall
  def method_missing(name, a, b)
    a+b
  end
end
```

la version avec `method_missing` s'exécute légèrement plus lentement. Sur ma machine, la première classe traditionnelle est environ 10 % plus rapide que si l'on appelle `add` sur la deuxième version où l'appel de la méthode inconnue transite par `method_missing`.

Un point plus important : l'usage excessif de `method_missing`, tout comme l'usage excessif de l'héritage, est un moyen très sûr de rendre votre code difficilement lisible. Lorsque vous employez la technique `method_missing`, vous créez des objets dont les messages sont traités plus ou moins par magie. Si quelqu'un lisait le code de notre système bancaire, il commencerait par chercher les méthodes `deposit` et `withdraw` dans nos classes proxy. Une personne qui maîtrise bien Ruby arriverait assez rapidement à la conclusion que vous utilisez la technique `method_missing`. Mais votre code ne devrait pas exiger plus de gymnastique mentale qu'il est nécessaire. Assurez-vous que vous avez une raison valable pour infliger cette difficulté au développeur qui arriverait après vous.

Proxies dans le monde réel

L'usage du pattern Proxy le plus populaire dans Ruby aujourd'hui est le proxy distant. À part le client SOAP de Ruby que j'ai mentionné ci-dessus, on trouve le package Distributed Ruby (drb). Il permet de développer en Ruby des applications distribuées,

reliées par un réseau TCP/IP. Le composant drb est très facile d'utilisation : quasiment tout objet typique Ruby peut agir en tant que service drb. Construisons un petit service tout bête :

```
class MathService
  def add(a, b) return
    a + b
  end
end
```

Pour exposer cet objet en tant que service drb il suffit d'écrire quelques lignes de code :

```
require 'drb/drb'
math_service = MathService.new
DRb.start_service("druby://localhost:3030", math_service)
DRb.thread.join
```

Nous créons une instance de `MathService` et murmurons l'incantation drb pour publier cet objet sur le port 3030. On peut maintenant démarrer le programme de service mathématique pour qu'il tourne en tâche de fond en attendant des requêtes entrantes.

Pour générer une requête il nous faut un programme client. Ouvrez une autre fenêtre ou déplacez-vous sur une autre machine et composez le code suivant. Tout d'abord, nous devons initialiser drb côté client :

```
require 'drb/drb'
DRb.start_service
```

Nous pouvons désormais appeler le service mathématique distant :

```
math_service = DRbObject.new_with_uri("druby://localhost:3030")
```

Il est clair que, si votre service est hébergé sur une autre machine ou un autre port, vous devrez adapter l'URL. L'exécution du programme client nous permet d'effectuer une opération mathématique incroyable :

```
sum = math_service.add(2,2)
```

Drb utilise effectivement le pattern Proxy car le service `math_service` du côté client est un proxy du service distant qui s'exécute à l'intérieur de l'interpréteur Ruby du côté serveur. Si vous jetez un œil dans le code de drb, vous trouverez la même technique fondée sur `method_missing` que celle étudiée dans ce chapitre.

En conclusion

Dans ce chapitre, nous avons examiné trois problèmes différents : protéger un objet contre l'accès non autorisé, cacher le fait que l'objet réside ailleurs sur le réseau, et

retarder autant que possible la création d'un objet coûteux. Il est remarquable que les trois problèmes aient une solution commune : le pattern Proxy. Dans le monde de la programmation, les proxies sont des imitateurs : il se font passer pour les autres objets. Un proxy conserve en interne une référence vers un vrai objet : cet objet est nommé sujet par le GoF.

Néanmoins, un proxy n'est pas seulement une passerelle pour des appels des méthodes d'un objet. Il sert de point intermédiaire entre le client et le sujet. "Cette opération est-elle autorisée ?" demande un proxy de contrôle d'accès. "Est-ce que cet objet réside réellement sur cette machine ?" demande un proxy distant. "Est-ce que j'ai déjà créé le sujet ?" demande un proxy virtuel. Bref, un proxy contrôle l'accès au sujet ! Nous avons également appris dans ce chapitre comment utiliser la technique `method_missing` pour réduire de manière significative l'effort de codage lorsqu'on développe des proxies.

Le pattern Proxy est le deuxième pattern que nous avons étudié où un objet prend la place d'un autre objet. Au Chapitre 9, nous avons vu le pattern Adapter, qui encapsule un objet dans un autre pour transformer l'interface du premier. Au premier abord, le pattern Proxy est semblable à Adapter : un objet se fait passer pour un autre. Mais Proxy ne change pas l'interface : l'interface de Proxy correspond exactement à celle de son sujet. Au lieu de modifier l'interface de l'objet encapsulé comme le fait un adaptateur, Proxy essaie de contrôler l'accès à cet objet.

Il s'avère que l'approche qui consiste à placer "un objet dans un autre" – une technique de conception à la poupée russe –, que nous avons rencontrée dans les patterns Adapter et Proxy, est tellement pratique que nous risquons fort de la voir resurgir avant la fin de ce livre. Pour être précis, elle réapparaîtra dès le chapitre suivant...

11

Améliorer vos objets avec Decorator

L'une des questions fondamentales du génie logiciel est la suivante : comment ajouter des fonctionnalités à un programme sans transformer l'ensemble en un bazar ingérable ? Jusqu'alors, nous avons appris comment diviser les détails de l'implémentation de vos objets en familles de classes à l'aide du pattern Template Method ainsi qu'à séparer des parties d'un algorithme avec le pattern Strategy. Nous savons également comment des objets doivent réagir à des requêtes entrantes avec le pattern Command ou comment rester au courant des modifications des autres objets avec le pattern Observer. Le pattern Composite et le pattern Itérateur nous aident chacun à leur manière à manipuler des collections d'objets.

Mais comment faire pour ajuster le niveau de responsabilité d'un objet pour qu'il soit capable de faire tantôt un peu plus et tantôt un peu moins ? Dans ce chapitre, nous allons étudier le pattern Decorator, qui permet d'ajouter des améliorations à un objet existant de façon simple. Ce pattern permet également d'empiler des couches de fonctionnalités afin de construire un objet muni des capacités optimales pour une situation donnée. Comme toujours, nous examinerons une alternative Ruby au pattern Decorator. Enfin, nous verrons pourquoi les objets qui portent beaucoup de décorations ne sont pas toujours idéals.

Décorateurs : un remède contre le code laid

Imaginez que vous ayez du texte stocké dans un fichier. La tâche va vous sembler assez primitive, mais supposons que votre système ait parfois besoin d'écrire du texte simple sans décor et qu'il faille parfois numéroter chaque ligne lorsqu'elle est écrite.

Il peut aussi s'agir d'ajouter une estampille à chaque ligne qui s'écrit dans le fichier mais aussi, pourquoi pas, de générer une somme de contrôle (checksum) pour s'assurer que le texte a été correctement sauvegardé.

Pour répondre à tous ces besoins, on pourrait partir d'un objet qui encapsule la classe `IO` de Ruby en y ajoutant des méthodes pour chaque variation de traitement :

```
class EnhancedWriter
  attr_reader :check_sum
  def initialize(path)
    @file = File.open(path, "w")
    @check_sum = 0
    @line_number = 1
  end

  def write_line(line)
    @file.print(line)
    @file.print("\n")
  end

  def checksumming_write_line(data)
    data.each_byte {|byte| @check_sum = (@check_sum + byte) % 256 }
    @check_sum += "\n"[0] % 256
    write_line(data)
  end

  def timestamping_write_line(data)
    write_line("#{Time.new}: #{data}")
  end

  def numbering_write_line(data)
    write_line("%{@line_number}: #{data}")
    @line_number += 1
  end

  def close
    @file.close
  end
end
```

On peut ensuite utiliser `EnhancedWriter` pour écrire soit du texte simple :

```
writer = EnhancedWriter.new('out.txt')
writer.write_line("A plain line")
```

soit une ligne avec la somme de contrôle :

```
writer.checksumming_write_line('A line with checksum')
puts("Checksum is #{writer.check_sum}")
```

soit une ligne avec une estampille ou avec un numéro de ligne :

```
writer.timestamping_write_line('with time stamp')
writer.numbering_write_line('with line number')
```

Il y a quelque chose qui ne va pas dans cette approche : tout ! Premièrement, tout client qui utilise `EnhancedWriter` serait obligé de savoir quel texte il écrit : numéroté, estampillé ou comprenant des sommes de contrôle. Le client ne peut pas prendre cette décision une fois à l'initialisation : cette information doit être disponible constamment, à l'écriture de chaque ligne. Si le client se trompe une fois – par exemple en utilisant `timestamping_write_line` au lieu de `numbering_write_line` ou en utilisant la méthode `write_line` basique alors qu'il lui faut appeler `checksumming_write_line` –, le nom de la classe `EnhancedIO` paraîtrait plutôt déplacé.

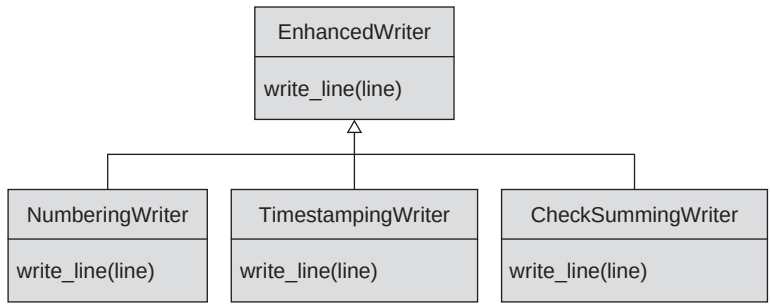
Un problème légèrement moins évident maintenant : tout est placé dans la même classe. La classe dans laquelle on mélange le code de numérotation avec le code de la somme de contrôle ainsi que la gestion de l'estampille est particulièrement sujet aux erreurs.

On pourrait séparer les responsabilités d'écriture en créant une classe parent et des sous-classes : autrement dit, recourir à notre vieil ami l'héritage selon le diagramme UML de la Figure 11.1.

Mais que faire si l'on veut générer une somme de contrôle des lignes numérotées ? Ou si on veut numéroter les lignes, mais pas avant d'ajouter une estampille ? Cela reste faisable, mais le nombre de classes devient ingérable, ce qui est clairement illustré à la Figure 11.2.

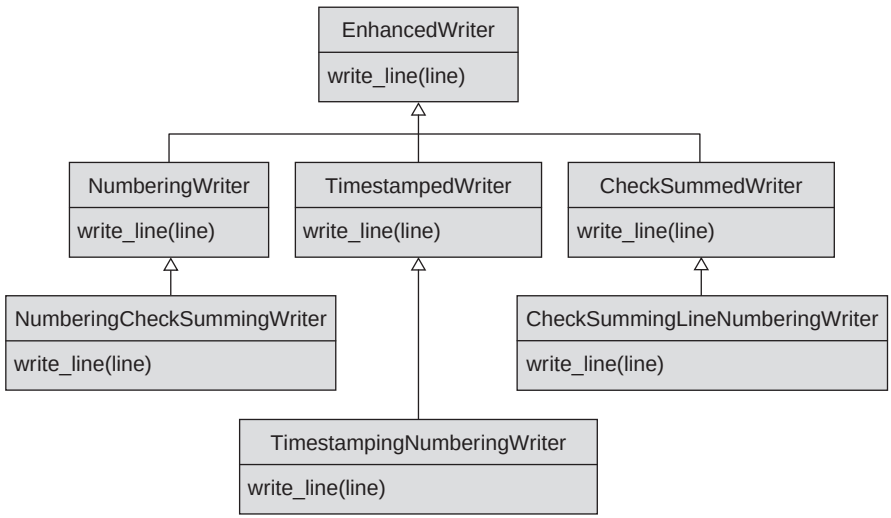
Figure 11.1

Résoudre le problème d'EnhancedWriter avec l'héritage



Il faut noter que, même avec cette foule de classes (voir Figure 11.2), on ne peut toujours pas afficher une somme de contrôle dans le texte estampillé après avoir numéroté les lignes. Le problème de l'approche fondée sur l'héritage est que toutes les combinaisons des fonctionnalités doivent être prises en compte dès la conception. Il est probable que vous ayez besoin non pas de toutes les combinaisons mais plutôt de quelques combinaisons précises.

Figure 11.2
L'héritage incontrôlé



Une solution plus adaptée consiste à se donner les moyens d'assembler les fonctionnalités nécessaires dynamiquement au moment de l'exécution. Commençons avec un objet basique capable d'écrire du texte simple et ajoutons quelques opérations de traitement supplémentaires :

```
class SimpleWriter
  def initialize(path)
    @file = File.open(path, 'w')
  end

  def write_line(line)
    @file.print(line)
    @file.print("\n")
  end

  def pos
    @file.pos
  end

  def rewind
    @file.rewind
  end

  def close
    @file.close
  end
end
```

Si vous voulez ajouter des numéros de lignes, insérez un objet (on peut le nommer par exemple `NumberingWriter`) entre la classe `SimpleWriter` et le client. Cet objet écrirait un numéro de ligne pour chacune des lignes et transférerait le résultat au `SimpleWriter`,

qui se chargerait de la sauvegarde sur le disque. `NumberingWriter` complète les capacités de `SimpleWriter` en le décorant, d'où le nom du pattern. Nous avons l'intention de définir plusieurs objets décorateurs, il faut donc factoriser le code générique dans une classe parent commune :

```
class WriterDecorator
  def initialize(real_writer)
    @real_writer = real_writer
  end

  def write_line(line)
    @real_writer.write_line(line)
  end

  def pos
    @real_writer.pos
  end

  def rewind
    @real_writer.rewind
  end

  def close
    @real_writer.close
  end
end

class NumberingWriter < WriterDecorator
  def initialize(real_writer)
    super(real_writer)
    @line_number = 1
  end

  def write_line(line)
    @real_writer.write_line("#{@line_number}: #{line}")
    @line_number += 1
  end
end
```

La classe `NumberingWriter` expose la même interface de base que `SimpleWriter`, or le client n'est pas obligé de se préoccuper du type de la classe à laquelle il a affaire : le fonctionnement de base des deux classes est identique.

Pour numéroter les lignes, il suffit d'insérer la classe `NumberingWriter` dans la chaîne du traitement après la classe `SimpleWriter` :

```
writer = NumberingWriter.new(SimpleWriter.new('final.txt'))
writer.write_line('Hello out there')
```

Le même patron s'applique pour développer un décorateur qui calcule les sommes de contrôle : un objet supplémentaire s'incruste entre le client et `SimpleWriter`, il calcule la somme des octets avant de transférer le résultat afin que `SimpleWriter` l'écrive dans un fichier :

```
class CheckSummingWriter < WriterDecorator
  attr_reader :check_sum
  def initialize(real_writer)
    @real_writer = real_writer
    @check_sum = 0
  end

  def write_line(line)
    line.each_byte {|byte| @check_sum = (@check_sum + byte) % 256 }
    @check_sum += "\n"[0] % 256
    @real_writer.write_line(line)
  end
end
```

La classe `CheckSummingWriter` diffère légèrement de notre premier décorateur car son interface est plus complexe. Cette classe expose la méthode `check_sum` en plus des autres méthodes typiques[1].

Enfin, nous pouvons définir la classe destinée à estampiller les données :

```
class TimeStampingWriter < WriterDecorator
  def write_line(line)
    @real_writer.write_line("#{Time.new}: #{line}")
  end
end
```

Cerise sur le gâteau : puisque les décorateurs partagent l'interface de base avec la classe initiale, rien ne nous oblige à fournir à nos décorateurs une instance de `SimpleWriter`. On a la liberté de leur passer un de nos décorateurs. Cela signifie que nous sommes en position de construire des chaînes de décorateurs à longueur illimitée afin que chacun d'eux puisse apporter sa contribution à l'ensemble. Il nous est enfin possible d'afficher une somme de contrôle du texte estampillé après l'avoir numéroté :

```
writer = CheckSummingWriter.new(TimeStampingWriter.new(
NumberingWriter.new(SimpleWriter.new('final.txt'))))
writer.write_line('Hello out there')
```

La décoration formelle

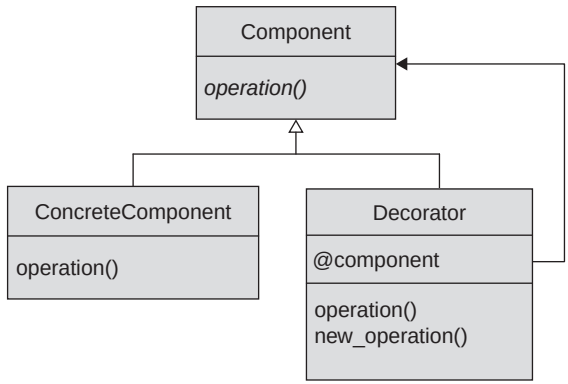
Tous les participants du pattern Decorator implémentent l'interface de `Component` (voir Figure 11.3).

La classe `ConcreteComponent` est l'objet "réel" qui implémente la fonctionnalité basique. Dans notre exemple, le rôle de `ConcreteComponent` est joué par `SimpleWriter`. La classe `Decorator` possède une référence vers `Component` – c'est-à-dire le `Component` suivant dans la chaîne des décorateurs – et elle implémente toutes les méthodes de la

1. La méthode `check_sum` a été générée par l'instruction `attr_reader`, bien entendu.

classe `Component`. Et on trouve trois classes `Decorator` : une pour numéroter des lignes, une pour calculer la somme de contrôle et une pour générer une estampille. Chacun des décorateurs ajoute une couche de fonctionnalités, en complétant au moins une méthode du composant de base avec ses propres capacités. Les décorateurs peuvent également introduire de nouvelles méthodes qui ne sont pas définies dans l'interface `Component`, mais ce comportement est facultatif. Dans notre exemple, le décorateur qui calcule les sommes de contrôle définit effectivement une nouvelle méthode.

Figure 11.3
Le pattern Decorator



Diminuer l'effort de délégation

Le pattern Decorator prend très au sérieux un des conseils du GoF : il s'appuie abondamment sur la délégation. Ce point est bien illustré par la classe `WriterDecorator`. Cette classe est constituée quasiment entièrement de code générique qui ne fait que déléguer à l'élément suivant de la chaîne.

On aurait pu éliminer tout ce code fastidieux avec une variation de la technique `method_missing` que nous avons apprise au Chapitre 10, mais le module `forwardable` serait probablement la solution la plus adaptée. Ce module génère automatiquement toutes les méthodes de délégation quasiment sans effort. Voici notre classe `WriterDecorator` réécrite pour profiter de `forwardable` :

```
require 'forwardable'
class WriterDecorator
extend Forwardable
  def_delegators :@real_writer, :write_line, :rewind, :pos, :close
  def initialize(real_writer)
    @real_writer = real_writer
  end
end
```

Le module `forwardable` fournit une méthode de classe `def_delegators`[1] qui accepte deux paramètres ou plus. Le premier argument est le nom de l'attribut d'instance[2]. Il est suivi par les noms d'une ou de plusieurs méthodes. La méthode `def_delegators` complète votre classe avec toutes les méthodes nommées. Chacune de ces nouvelles méthodes délègue à l'objet indiqué par l'attribut. La classe `WriterDecorator` se retrouve donc avec les méthodes `write_line`, `rewind`, `pos` et `close`, qui délèguent le travail à `@real_writer`.

Le module `forwardable` apporte plus de précision par rapport à la technique `method_missing` car il vous laisse le choix des méthodes que vous souhaitez déléguer. On peut certainement insérer la logique de délégation de quelques méthodes dans `method_missing`, mais la vraie force de cette technique réside dans sa capacité à déléguer un grand nombre d'appels.

Les alternatives dynamiques au pattern Decorator

La flexibilité de Ruby au moment de l'exécution présente certaines alternatives intéressantes au pattern Decorator du GoF. Il est possible de bénéficier de la plupart des avantages des décorateurs soit à l'aide de l'encapsulation dynamique des méthodes soit en décorant vos objets avec des modules.

Les méthodes d'encapsulation

Nous avons déjà appris que Ruby permet de modifier le comportement d'une instance ou d'une classe entière quasiment à tout instant. Fort de cette flexibilité et du mot clé `alias`, on peut doter `SimpleWriter` de la fonctionnalité requise pour ajouter une estampille :

```
w = SimpleWriter.new('out')
class << w
  alias old_write_line write_line
  def write_line(line)
    old_write_line("#{Time.new}: #{line}")
  end
end
```

1. Vous avez probablement remarqué que dans `WriterDecorator` nous étendons le module `Forwardable` au lieu de l'inclure. La différence est subtile : le module `Forwardable` essaie d'ajouter des méthodes de classe et non pas des méthodes d'instance.
2. Accompagné de "@" pour une raison étrange.

Le mot clé `alias` crée un nouveau nom pour une méthode existante. Dans le code ci-dessus on commence par déclarer un `alias` pour la méthode initiale `write_line`, afin qu'elle puisse être appelée `write_line` aussi bien que `old_write_line`. Ensuite, on redéfinit la méthode `write_line`, mais la méthode `old_write_line` continue à pointer vers la définition initiale, ce qui est primordial. Le reste est facile : la nouvelle méthode estampille chacune des lignes et appelle la méthode initiale (maintenant nommée `old_write_line`), qui à son tour écrit le texte estampillé dans un fichier.

Heureusement pour vous, futurs décorateurs potentiels, la technique d'encapsulation de méthode présente certaines limites. La collision de noms, notamment, représente un des dangers. Par exemple, si l'on voulait numéroter nos lignes deux fois avec le code actuel, on perdrait la référence vers la méthode `write_line` initiale à cause du double `alias`. On pourrait probablement inventer une convention ingénieuse pour éviter la collision de noms mais, lorsque vos décorations gagnent en complexité, la nécessité de les déplacer dans leurs propres classes devient de plus en plus une évidence. Toutefois, la technique d'encapsulation de méthodes reste utile pour des cas simples. Elle mérite d'être maîtrisée par tout développeur Ruby.

Décorer à l'aide de modules

Un autre moyen d'injecter des fonctionnalités dans un objet Ruby consiste à rajouter dynamiquement des modules à l'aide de la méthode `extend`. Pour appliquer cette technique il faut refactoriser notre code et transformer nos classes décorateurs en modules :

```
module TimeStampingWriter
  def write_line(line)
    super("#{Time.new}: #{line}")
  end
end

module NumberingWriter
  attr_reader :line_number
  def write_line(line)
    @line_number = 1 unless @line_number
    super("#{@line_number}: #{line}")
    @line_number += 1
  end
end

class Writer
  def write_line(line)
    @f.write_line(line)
  end
end
```

La méthode `extend` insère un module dans l'arbre de l'héritage d'un objet avant sa classe normale. On peut donc partir d'un objet `SimpleWriter` et ensuite simplement ajouter la fonctionnalité requise :

```
w = SimpleWriter.new('out')
w.extend(NumberingWriter)
w.extend(TimeStampingWriter)
w.write_line('hello')
```

Le dernier module ajouté est le premier appelé. Dans l'exemple ci-dessus le flux de traitement commence par `TimeStampingWriter`, continue avec `NumberingWriter` et s'achève par `Writer`.

Les deux techniques dynamiques fonctionnent bien. En réalité, ce sont les solutions de prédilection dans le code Ruby existant. Mais il existe un défaut inhérent aux deux approches : il est difficile d'annuler la décoration. En effet, extraire une méthode encapsulée est assez fastidieux et il est tout simplement impossible d'exclure un module déjà inclus.

User et abuser du pattern Decorator

Le pattern Decorator classique est adoré des développeurs et moins apprécié par leurs clients. Nous avons appris que le pattern Decorator aide les programmeurs à séparer de manière propre les différentes responsabilités – extraire la numérotation de lignes dans une classe, les sommes de contrôle dans une deuxième et l'estampille dans une troisième. Le moment de vérité survient lorsque quelqu'un tente de rassembler toutes les briques dans un ensemble fonctionnel. Le client est obligé de recoller toutes les pièces seul au lieu de pouvoir instancier un objet unique, par exemple `EnhancedWriter.new(path)`. Évidemment, il y a des solutions qui permettent de simplifier la tâche d'assemblage. S'il existe des chaînes de décorateurs dont vos clients ont besoin fréquemment, n'hésitez pas à leur fournir un utilitaire (peut-être un Builder[1]?) d'assemblage des chaînes de traitement.

Un autre point à retenir lorsqu'on implémente le pattern Decorator, c'est que l'interface du `Component` doit rester simple. Il faut éviter de rendre l'interface excessivement complexe, car cette complexité sera un obstacle à une implémentation correcte de chacun des décorateurs.

1. Voir le Chapitre 14.

La baisse de performance liée à une longue chaîne de décorateurs est un autre inconvénient potentiel du pattern Decorator. Lorsque vous remplacez la classe monolithique `ChecksummingNumberingTimestampingWriter` par une chaîne de décorateurs, vous gagnez énormément en séparation de responsabilités et en clarté de code. Le prix à payer tient à la multiplication des objets sollicités par votre programme. Ce n'est pas bien grave lorsque, comme dans notre exemple, on manipule quelques fichiers ouverts. Cela peut le devenir lorsqu'il faut gérer chaque salarié d'une grande entreprise. Souvenez-vous que mis à part le nombre des objets impliqués, les données que vous envoyez à travers la chaîne doivent passer par n décorateurs et changer n fois de propriétaire : du premier décorateur au deuxième, ensuite au troisième, etc.

Pour finir, la technique d'encapsulation de méthodes présente l'inconvénient d'être difficile à déboguer. Souvenez-vous que vos méthodes apparaîtront dans la pile d'appels avec des noms qui ne correspondent pas à leurs noms dans les fichiers source. Ce n'est pas rédhibitoire mais c'est un point supplémentaire dont il faut tenir compte.

Les décorateurs dans le monde réel

On trouve un bon exemple du style de décoration fondé sur l'encapsulation de méthode dans ActiveSupport, le module d'assistants (*helpers*) employé par Rails. ActiveSupport ajoute à tous les objets une méthode nommée `alias_method_chain`. Cette méthode vous permet de décorer vos méthodes avec un nombre de fonctionnalités illimité. Pour se servir d'`alias_method_chain`, on commence par une méthode simple dans votre classe, par exemple `write_line` :

```
def write_line(line)
  puts(line)
end
```

Ensuite, on ajoute une deuxième méthode qui insère une certaine décoration à la méthode initiale :

```
def write_line_with_timestamp(line)
  write_line_without_timestamp("#{Time.new}: #{line}")
end
```

Enfin, on appelle `alias_method_chain` :

```
alias_method_chain :write_line, :timestamp
```

La méthode `alias_method_chain` renomme la méthode initiale `write_line` en `write_line_without_timestamp`, alors que la méthode `write_line_with_timestamp` est renommée en `write_line`, ce qui crée une chaîne de méthodes. L'avantage

d'`alias_method_chain`, comme l'indique son nom, tient à la possibilité de chaîner un certain nombre de méthodes de décoration. Par exemple, on peut aussi compléter l'ensemble avec une méthode de numérotation de lignes :

```
def write_line_with_numbering(line)
  @number = 1 unless @number
  write_line_without_numbering("#{@number}: #{line}")
  @number += 1
end
alias_method_chain :write_line, :numbering
```

En conclusion

Le pattern Decorator est une technique simple qui permet d'assembler au moment de l'exécution la fonctionnalité requise. Cette approche est une alternative à la création d'un objet monolithique supportant toutes les fonctionnalités possibles avec d'innombrables classes et sous-classes couvrant toutes les combinaisons de fonctions possibles et imaginables. L'idée du pattern Decorator consiste à compléter une classe délivrant la fonctionnalité de base à l'aide de plusieurs décorateurs. Chacun des décorateurs supporte la même interface de base mais apporte sa contribution à la fonctionnalité. Les décorateurs acceptent un appel de méthode, exécutent leur fonction unique et font suivre l'appel au composant suivant. La clé de l'implémentation du pattern Decorator est le fait que le composant suivant peut être un autre décorateur prêt à apporter sa contribution, mais ce peut être également l'objet réel qui finalise la requête.

Le pattern Decorator vous permet de commencer par une fonctionnalité basique et d'empiler des fonctions supplémentaires, un décorateur après l'autre. Puisque le pattern Decorator construit des couches de fonctionnalités au moment de l'exécution, vous êtes libre de choisir la combinaison adaptée à vos besoins.

Le pattern Decorator est le dernier pattern du type "un objet en remplace un autre" que nous étudions dans ce livre. Le premier était le pattern Adapter, qui encapsule un objet à l'interface inadaptée dans un autre objet qui expose l'interface requise. Le deuxième était le pattern Proxy. Il encapsule également un objet, mais sans changer son interface. L'interface d'un proxy est identique à celle du sujet. Un proxy existe pour contrôler, non pas pour traduire. Les proxies sont pratiques pour contrôler la sécurité, cacher le fait que le vrai objet réside sur le réseau ou pour retarder autant que possible la création d'un vrai objet. Enfin, nous avons vu dans ce chapitre le pattern Decorator, qui permet d'ajouter des fonctionnalités variables à un objet basique.

12

Créer un objet unique avec Singleton

Pauvre pattern Singleton ! Même les programmeurs qui ne savent rien sur les patterns connaissent le pattern Singleton. Généralement, ils savent une chose : les singletons, c'est Mal, avec un "M" majuscule. Et, pourtant, il est difficile de s'en passer. Les singletons sont partout. Dans le monde Java on les trouve dans les logiciels les plus répandus, par exemple Tomcat et JDOM. Pour vous donner encore quelques exemples, on rencontre des singletons Ruby dans WEBrick, rake et même Rails. Quelles caractéristiques du pattern Singleton le rendent si indispensable et pourtant si largement détesté ? Dans les pages qui suivent, nous verrons pourquoi les singletons sont si utiles et comment s'y prendre pour les développer correctement. Nous allons découvrir comment construire des singletons et des presque-singletons en Ruby, pourquoi les singletons peuvent devenir une source de problèmes et quelles mesures prendre pour diminuer ces difficultés.

Objet unique, accès global

La motivation sous-jacente du pattern Singleton est très simple : certaines choses sont uniques. Par exemple, les programmes ont souvent un seul fichier de configuration ou bien un seul fichier de log. Les applications munies d'une interface graphique comprennent fréquemment une fenêtre principale et reçoivent les données à partir d'exactement un clavier. Pour finir, on peut citer l'exemple de nombreuses applications qui communiquent avec exactement une base de données. Si vous avez une seule instance d'une classe et que beaucoup de code doive y accéder, il paraît injustifié de passer l'objet d'une méthode à l'autre. Dans cette situation, les membres du GoF suggèrent de développer un singleton, une classe qui propose un accès global à une seule et unique instance.

Il existe plusieurs façons de reproduire le comportement singleton en Ruby et nous allons commencer par la méthode la plus proche de celle recommandée par le GoF : laissons la classe de l'objet singleton gérer la création et l'accès à son instance unique. Pour y arriver, jetons un œil sur les variables de classe et les méthodes de classe en Ruby.

Variables et méthodes de classe

Jusqu'alors, tout le code que nous avons écrit se fondait sur des méthodes et variables d'instance, le code et les données étaient attachés à des instances particulières. Tout comme la majorité des langages orientés objet, Ruby supporte également des variables et méthodes attachées à une classe[1].

Variables de classe

Comme mentionné plus haut, une variable de classe est une variable attachée à une classe[2] plutôt qu'à une instance de la classe. Créer une variable de classe est très simple : il suffit d'ajouter au nom de la variable une arobase supplémentaire (`@`). Voici un exemple d'une classe qui compte le nombre des appels de la méthode `increment` dans deux variables différentes : une variable d'instance et une variable de classe.

```
class ClassVariableTester
  @@class_count = 0
  def initialize
    @instance_count = 0
  end

  def increment
    @@class_count = @@class_count + 1
    @instance_count = @instance_count + 1
  end

  def to_s
    "class_count: #{@@class_count} instance_count: #{@instance_count}"
  end
end
```

1. De nombreux autres langages, notamment C++ et Java, utilisent le nom "statique" pour désigner les méthodes et variables de classe. Malgré la différence terminologique, le principe est le même.
2. En réalité, des variables de classe en Ruby sont attachées à la hiérarchie entière de l'héritage. Par conséquent, une classe partage l'ensemble des variables de classe avec sa classe mère ainsi que toutes ses sous-classes. La plupart des programmeurs Ruby considèrent ceci comme une caractéristique étrange du langage. Des discussions sont en cours pour modifier cette fonctionnalité.

On peut créer une instance `ClassVariableTester` et appeler sa méthode `increment` deux fois :

```
cl = ClassVariableTester.new
cl.increment
cl.increment
puts("c1: #{c1}")
```

Il n'est pas étonnant que les deux compteurs contiennent la valeur 2 :

```
cl: class_count: 2 instance_count: 2
```

Les choses deviennent plus intéressantes lorsque vous créez une deuxième instance de cette classe :

```
c2 = ClassVariableTester.new
puts("c2: #{c2}")
```

Le résultat est

```
c2: class_count: 2 instance_count: 0
```

La différence s'explique par le fait que le compteur d'instance du deuxième objet `ClassVariableTester` a été réinitialisé à zéro alors que le compteur de classe s'est incrémenté normalement.

Méthodes de classe

Créer des méthodes de classe en Ruby est à peine plus compliqué. Nous ne pouvons pas simplement créer une classe et définir une méthode :

```
class SomeClass
  def a_method
    puts('hello from a method')
  end
end
```

Nous savons déjà que ce genre de code crée des méthodes d'instances :

```
SomeClass.a_method
instance.rb:11: undefined method 'a_method' for SomeClass:Class
```

Le secret de la définition d'une méthode de classe est de savoir quand on se trouve à l'intérieur de la définition d'une classe, mais à l'extérieur de la définition d'une méthode. La classe courante est référencée par la variable `self`. Vous n'êtes pas obligé de me croire sur parole. Si l'on exécute le code suivant :

```
class SomeClass
  puts ("Inside a class def, self is #{self}")
end
```

voici le résultat obtenu :

```
Inside a class def, self is SomeClass
```

Cette information nous aidera à définir une méthode de classe :

```
class SomeClass
  def self.class_level_method
    puts('hello from the class method')
  end
end
```

Nous pouvons désormais appeler la méthode `class_level_method` au niveau de la classe, comme le suggère son nom :

```
SomeClass.class_level_method
```

Si la syntaxe `self.method_name` ne vous plaît pas, Ruby propose une autre option. On peut définir une méthode de classe en déclarant son nom explicitement :

```
class SomeClass
  def SomeClass.class_level_method
    puts('hello from the class method')
  end
end
```

En ce qui concerne le choix syntaxique, les programmeurs Ruby semblent être partagés en deux groupes égaux : certains préfèrent `self`, les autres préfèrent le nom explicite de la classe. Personnellement, j'aime le format `self` car il faut faire moins de modifications si l'on renomme la classe ou si le code est transplanté dans une autre classe.

Première tentative de création d'un singleton Ruby

Maintenant que nous savons créer des variables et méthodes de classes, nous avons tous les outils pour définir un singleton. Partons d'une classe ordinaire pour la transformer en un singleton. Supposons que vous ayez une classe de log chargée de collecter les messages émis par votre programme. La version non singleton de votre classe de log pourrait ressembler à ceci :

```
class SimpleLogger
  attr_accessor :level
  ERROR = 1
  WARNING = 2
  INFO = 3
  def initialize
    @log = File.open("log.txt", "w")
    @level = WARNING
  end
```

```
  def error(msg)
    @log.puts(msg)
    @log.flush
  end

  def warning(msg)
    @log.puts(msg) if @level >= WARNING
    @log.flush
  end

  def info(msg)
    @log.puts(msg) if @level >= INFO
    @log.flush
  end
end
```

On pourrait créer une instance de cette classe et la passer partout :

```
logger = SimpleLogger.new
logger.level = SimpleLogger::INFO
logger.info('Doing the first thing')
# Effectuer la première opération...
logger.info('Now doing the second thing')
# Effectuer la deuxième opération...
```

Gestion de l'instance unique

Le seul but du pattern Singleton est d'éviter de passer un objet tel que l'objet de log partout dans le programme. Il serait judicieux de confier la gestion de l'instance unique à la classe `SimpleLogger`. Mais comment transformer `SimpleLogger` en un singleton ? Tout d'abord, on crée une variable de classe pour contenir la seule et unique instance de la classe. Une méthode de classe serait indispensable pour retourner l'instance singleton :

```
class SimpleLogger
  # Beaucoup de code supprimé...
  @@instance = SimpleLogger.new
  def self.instance
    return @@instance
  end
end
```

Quel que soit le nombre d'appels vers la méthode `SimpleLogger`, elle retournerait toujours le même objet de log :

```
logger1 = SimpleLogger.instance   # Retourne l'objet de log
logger2 = SimpleLogger.instance   # Retourne exactement le même objet
                                  # de log
```

Il est très pratique de pouvoir accéder au singleton de log à tout moment pour écrire des messages :

```
SimpleLogger.instance.info('Computer wins chess game.')
SimpleLogger.instance.warning('AE-35 hardware failure predicted.')
```

```
SimpleLogger.instance.error(
  ➥'HAL-9000 malfunction, take emergency action!')
```

S'assurer de l'unicité

Notre singleton est opérationnel, mais il n'est pas complet. Souvenez-vous d'une des exigences des singletons : il faut s'assurer que l'objet singleton est l'instance unique de sa classe. Jusqu'alors, nous avons ignoré cette exigence. En l'état, tout programme peut appeler `SimpleLogger.new` pour créer une deuxième instance de notre "singleton". Comment faire pour protéger la classe `SimpleLogger` contre des instanciations non sollicitées ?

Il suffit de rendre privée la méthode `new` de `SimpleLogger` :

```
class SimpleLogger
# Beaucoup de code supprimé...
@@instance = SimpleLogger.new
  def self.instance
    return @@instance
  end
  private_class_method :new
end
```

Deux points sont à noter dans le fragment de code précédent : le premier n'est qu'un détail, et le deuxième est plus profond. En ajoutant l'instruction `private_class_method` nous avons rendu la méthode `new` de la classe privée et par conséquent inaccessible. Aucune autre classe ne pourrait créer des instances de notre logger : c'est un détail. Il faut noter le point plus global : la méthode `new` n'est qu'une méthode de classe classique. Elle exécute effectivement des actions magiques et invisibles pour allouer l'objet, mais finalement c'est une méthode de classe comme une autre.

Le module Singleton

Notre singleton est maintenant prêt : nous avons rempli toutes les exigences définies par le GoF pour l'implémentation des singletons. Notre classe instancie un objet unique, tout code intéressé peut accéder à cette instance et nul ne peut créer une deuxième instance.

Néanmoins, notre implémentation du pattern Singleton présente un défaut. Que se passe-t-il si l'on veut créer une deuxième classe singleton, pour nos données de configuration, par exemple ? Nous allons devoir recommencer l'exercice : créer une variable de classe pour l'instance de singleton ainsi qu'une méthode de classe pour y accéder. Et n'oubliez pas de rendre la nouvelle méthode `new` privée. Et lorsqu'une troisième instance devient nécessaire, il faudra recommencer à nouveau. Finalement, beaucoup d'efforts répétés.

Heureusement, il est possible d'éviter ce travail. Au lieu de prendre la peine de transformer nos classes en singletons à la main, on peut simplement inclure le module `Singleton` :

```
require 'singleton'
class SimpleLogger
include Singleton
  # Beaucoup de code supprimé...
end
```

Le module `Singleton` réalise le plus gros du travail : il définit une variable de classe et l'initialise avec l'instance unique, il injecte également la méthode de classe pour retourner l'instance et rend la méthode `new` privée. Il ne nous reste qu'à inclure le module. Vu de l'extérieur, le nouvel objet de log fondé sur le module `Singleton` est identique à l'implémentation faite à la main : il suffit d'appeler `SimpleLogger.instance` pour récupérer son instance et le tour est joué.

Singletons à instanciation tardive ou immédiate

Il existe une différence significative entre notre propre implémentation de singleton et celle fournie par le module `Singleton`. Souvenez-vous que notre implémentation crée l'instance de singleton lorsque la classe est définie :

```
class SimpleLogger
  # Beaucoup de code supprimé...
  @@instance = SimpleLogger.new
  # Beaucoup de code supprimé...
end
```

Par conséquent, notre instance de singleton est créée avant même que le code client ait la possibilité d'appeler `SimpleLogger.instance`. La création de l'instance du singleton avant qu'elle ne soit nécessaire s'appelle l'instanciation immédiate. Contrairement à notre approche, le module `Singleton` attend un appel à l'instance avant de la créer réellement. Cette technique est connue sous le nom d'instanciation tardive.

Alternatives au singleton classique

La technique de développement des singletons gérés par une classe employée précédemment reflète fidèlement l'implémentation recommandée par le GoF. Mais il existe beaucoup d'autres variantes pour implémenter le comportement au singleton. Voici quelques alternatives auxquelles on pourrait recourir pour atteindre le même effet.

Variables globales en tant que singletons

Par exemple, on pourrait utiliser une variable globale en tant que singleton. Je fais une pause pour laisser les cris d'horreur se calmer... En Ruby, toute variable dont le nom commence par le caractère `$` – tel que `$logger` – est globale. L'accès global vers un singleton est donc facilement réalisable à l'aide d'une variable globale : quel que soit le contexte – classe, module ou méthode –, `$logger` reste accessible et retourne toujours

le même objet. Puisqu'il n'existe qu'une instance de la variable globale donnée et compte tenu de son accès universel, une variable globale paraît une bonne plate-forme pour implémenter des singletons.

Hélas, il manque aux variables globales une des caractéristiques fondamentales des singletons ! La variable `$logger` conserve une référence à un objet unique, mais il est impossible de contrôler sa valeur. On pourrait commencer par notre pseudo-singleton global :

```
$logger = SimpleLogger.new
```

Mais rien ne peut empêcher la modification de la valeur par du code malveillant :

```
$logger = LoggerThatDoesSomethingBad.new
```

Si les modifications posent un problème, nous devrions peut-être nous tourner vers un autre type de variable Ruby. Les constantes sont des variables qui ont non seulement une portée globale, mais qui sont également résistantes aux changements. Souvenez-vous qu'une constante en Ruby est une variable dont le nom commence par une lettre majuscule et dont la valeur est censée rester inchangée :

```
Logger = SimpleLogger.new
```

Nous avons mentionné au Chapitre 2 que Ruby déclenche un avertissement lorsque quelqu'un change la valeur d'une constante. C'est une amélioration par rapport à l'attitude "tout est possible" des variables globales. Mais est-ce une solution simple pour implémenter un singleton ?

Pas vraiment. Les variables globales et les constantes partagent un certain nombre d'inconvénients. Premièrement, si l'on choisit une variable globale ou une constante, il n'existe aucun moyen de retarder l'instanciation de l'objet singleton jusqu'au moment où il devient nécessaire. Une variable globale ou une constante sont présentes dès leur initialisation. Deuxièmement, rien dans ces techniques n'empêche la création d'une deuxième ou troisième instance de votre "singleton". On pourrait essayer de gérer ce problème séparément. Par exemple, on pourrait créer une instance de singleton et ensuite modifier la classe pour qu'elle refuse d'instancier de nouveaux objets, mais ce genre de solution ne semble pas être très propre.

Étant donné que les variables globales et les constantes ne font pas l'affaire, quelles autres voies nous reste-t-il pour implémenter des singletons ?

Des classes en tant que singletons

Comme nous l'avons appris, des méthodes ainsi que des variables peuvent être définies directement sur l'objet classe. En effet, notre implémentation initiale de ce pattern faisait appel à des méthodes et à des variables de classe pour gérer l'instance de l'objet

singleton. Mais puisque nous pouvons avoir des méthodes et des variables attachées à une classe pourquoi ne pas utiliser la classe même en tant que conteneur de la fonctionnalité singleton ? Chaque classe est unique, il ne peut exister qu'un seul `SimpleLogger` chargé à l'instant donné. On pourrait donc définir notre fonctionnalité singleton à l'aide des méthodes et des variables de l'objet classe :

```
class ClassBasedLogger
  ERROR = 1
  WARNING = 2
  INFO = 3
  @@log = File.open('log.txt', 'w')
  @@level = WARNING
  def self.error(msg)
    @@log.puts(msg)
    @@log.flush
  end

  def self.warning(msg)
    @@log.puts(msg) if @@level >= WARNING
    @@log.flush
  end

  def self.info(msg)
    @@log.puts(msg) if @@level >= INFO
    @@log.flush
  end

  def self.level=(new_level)
    @@level = new_level
  end

  def self.level
    @@ievel
  end
end
```

L'utilisation de l'objet singleton fondé sur une classe n'est pas difficile :

```
ClassBasedLogger.level = ClassBasedLogger::INFO
ClassBasedLogger.info('Computer wins chess game.')
ClassBasedLogger.warning('AE-35 hardware failure predicted.')
ClassBasedLogger.error('HAL-9000 malfunction, take emergency action!')
```

La technique "classe en tant que singleton" offre un avantage clé par rapport aux variables globales et aux constantes : on est sûr que la création d'une deuxième instance est impossible. Toutefois, avec cette technique l'initialisation tardive présente un problème. Votre classe est initialisée lors du chargement (typiquement lorsque quelqu'un inclut le fichier où réside la classe avec l'instruction `require`). Donc, on ne contrôle pas le moment de son initialisation. Un autre défaut de cette technique tient au fait que programmer des variables et des méthodes de classe n'est pas aussi facile que

coder des méthodes et des variables d'instance traditionnelles ; on a un sentiment étrange quand on écrit tous ces `self.methods` et ces `@@variables`.

Des modules en tant que singletons

Utiliser un module pour encapsuler le comportement singleton est une autre possibilité. On a déjà noté dans ce chapitre que les modules ont beaucoup de points communs avec les classes. En effet, les modules ressemblent tellement aux classes qu'on peut définir des méthodes et variables au niveau des modules comme on le faisait avec des classes. Mise à part la déclaration `module`, l'implémentation fondée sur un module est identique à celle fondée sur une classe :

```
module ModuleBasedLogger
ERROR = 1
WARNING = 2
INFO = 3
@@log = File.open("log.txt", "w")
@@level = WARNING
  def self.error(msg)
    @@log.puts(msg)
    @@log.flush
  end
  # Beaucoup de code identique à
  # ClassBasedSingleton supprimé...
end
```

Tout comme les méthodes de classe, les méthodes de module sont universellement accessibles :

```
ModuleBasedLogger.info('Computer wins chess game.')
```

La technique "module en tant que singleton" possède un avantage considérable par rapport à l'approche "classe en tant que singleton". Puisqu'un module ne peut pas être instancié (ce qui est la différence clé entre une classe et un module), le but d'un singleton fondé sur un module se manifeste plus clairement à la lecture du code : voici un ensemble de méthodes destinées à être appelées mais dont on ne peut rien instancier.

Ceinture de sécurité ou carcan ?

Le débat sur les moyens alternatifs d'implémentation du pattern Singleton soulève certaines questions concernant les fonctions de sécurité incorporées dans le langage et l'impact qu'elles peuvent avoir dans un langage aussi flexible que Ruby. Par exemple, nous avons vu que le fait d'incorporer le module Singleton provoque le changement de modificateur d'accès de la méthode `new` en `private`. Ceci empêche la création d'une deuxième ou troisième instance de la classe singleton. Si notre classe est définie de la façon suivante :

```
require 'singleton'
class Manager
  include Singleton
  def manage_resources
    puts("I am managing my resources")
  end
end
```

On ne peut pas créer une autre instance de `Manager`. Par exemple, si l'on essaie

```
m = Manager.new
```

on obtient

```
private method 'new' called for Manager:Class
```

En vérité, le module `Singleton` ne peut pas empêcher une telle action. Pour contourner l'avertissement, il suffit de bien comprendre le fonctionnement de `Singleton` et d'avoir une idée de la mécanique de la méthode `public_class_method` (le frère ennemi de `private_class_method`) :

```
class Manager
  public_class_method :new
end
m = Manager.new
```

Nous avons noté ci-dessus que l'avantage des singletons fondés sur un module ou sur une classe réside dans l'impossibilité de créer une deuxième instance. Si l'appel est lancé par hasard, c'est effectivement impossible. Mais peu importe quelle classe vous utilisez, que ce soit `ClassBasedLogger` ou son cousin `ModuleBasedLogger`, ce ne sont que des objets. Et tous les objets Ruby héritent de la méthode `clone`. Cette méthode est un moyen formidable pour contourner la protection des singletons que nous avons établie avec tant d'effort :

```
a_second_logger = ClassBasedLogger.clone
a_second_logger.error('using a second logger')
```

On peut en effet surcharger la méthode `clone` dans `ClassBasedLogger` pour éviter le clonage non sollicité, mais une personne déterminée pourrait aussi bien rouvrir la classe et supprimer la protection.

Le but est de ne pas encourager ce type de code, mais d'illustrer que dans un langage où quasiment tout ce qui est fait au moment de l'exécution peut aussi être défait dynamiquement, très peu de décisions sont définitives. La philosophie Ruby est telle que, si vous décidez de contourner l'intention très claire de l'auteur de la classe `ClassBasedLogger` et de la cloner, le langage vous donne les moyens de le faire. La décision est alors prise sciemment par le développeur et non pas par le langage. Ruby ouvre

quasiment tout à la modification, vous êtes donc libre de choisir vos propres pratiques et c'est donc à vous de faire les bons choix.

User et abuser du pattern Singleton

Maintenant que nous savons implémenter des singletons, essayons de comprendre pourquoi c'est le pattern probablement le plus détesté.

Ce sont simplement des variables globales, n'est-ce pas ?

Commençons par le problème le plus évident : un singleton ressemble fortement à son cousin hors la loi, la variable globale. Peu importe le type de singleton que vous implémentez : un objet géré par une classe préconisé par le GoF ou un ensemble de méthodes et de variables attachées à une classe ou à un module, vous créez un objet unique avec la portée globale. Un singleton ouvre la porte secrète par laquelle des parties complètement indépendantes de votre programme peuvent communiquer, ce qui augmente fortement le couplage entre des composants de votre système. Les conséquences horribles de ce couplage expliquent pourquoi les ingénieurs logiciel ont abandonné l'usage des variables globales.

Il n'existe qu'une seule solution au problème : ne faites pas cela. Lorsqu'ils sont utilisés correctement, les singletons ne sont pas des variables globales. Leur but est de modéliser des choses qui n'ont qu'une seule occurrence. Puisque l'occurrence est unique, on peut utiliser un singleton en tant que canal unique de communication entre différentes parties de votre programme. Mais ne faites pas cela. Les singletons ne sont pas différents des autres patterns : si vous en abusez, vous pouvez provoquer beaucoup de dégâts. Je ne peux que répéter cette recommandation : ne faites jamais cela.

Vous en avez combien, des singletons ?

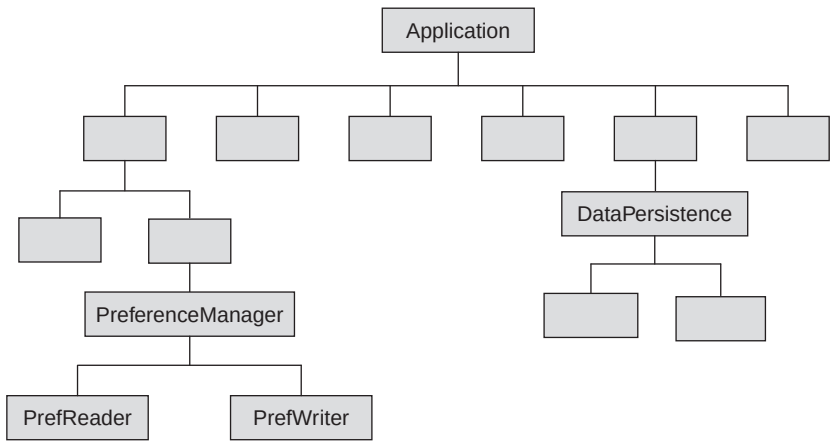
La deuxième façon d'avoir des problèmes avec le pattern Singleton peut paraître évidente mais elle est très courante : définir des singletons en grande quantité. Lorsque vous considérez l'utilisation du pattern Singleton, posez-vous cette question : est-ce que je suis sûr que cette chose est unique ? Le pattern Singleton nous fournit un moyen de créer un modèle d'une instance unique, mais ce modèle est accompagné d'une fonction très pratique qui rend cette instance facilement accessible, il suffit d'appeler `SimpleLogger.instance`. La simplicité d'accès peut avoir un effet hypnotique : "Mon code serait beaucoup plus simple si cette chose était un singleton." N'écoutez pas ce chant des sirènes. Concentrez-vous sur la question ci-dessus et considérez l'accès facile comme un bonus.

Singletons pour les intimes

Largement propager l'information sur le fait qu'un objet est un singleton est une autre erreur répandue. On peut voir le fait qu'une classe est un singleton comme un simple détail d'implémentation : une fois récupéré l'objet fichier de configuration, peu vous importe de savoir comment cette opération est implémentée. Souvenez-vous que l'on peut retrouver l'objet singleton à tout endroit du code et ensuite le passer comme un objet classique.

Cette technique peut être pratique lorsque votre application doit utiliser un singleton à plusieurs endroits du code. Par exemple, votre application pourrait être structurée selon le diagramme de la Figure 12.1.

Figure 12.1
Une application utilisant des singletons dans des composants indépendants



Imaginez que la classe `PreferenceManager` ainsi que les classes qu'elle utilise nécessitent un accès à la base de données. C'est aussi le cas de la classe `DataPersistence` et de ses amis.

Ensuite, imaginez que l'application fasse appel à une instance unique de la classe `DatabaseConnectionManager` pour gérer toute connexion à la base. Vous décidez d'implémenter `DatabaseConnectionManager` comme un singleton :

```
require 'singleton'
class DatabaseConnectionManager
include Singleton
  def get_connection
    # Retourner la connexion à la base de données...
  end
end
```

Question : quelles classes sont au courant que `DatabaseConnectionManager` est un singleton ? On pourrait rendre cette information largement disponible, accessible aux objets `PrefReader` et `PrefWriter` :

```
class PreferenceManager
  def initialize
    @reader = PrefReader.new
    @writer = PrefWriter.new
    @preferences = { :display_splash => false,
    ➥:background_color => :blue }
  end

  def save_preferences
    preferences = {}
    @writer.write(@preferences)
  end

  def get_preferences
    @preferences = @reader.read
  end
end

class PrefWriter
  def write(preferences)
    connection = DatabaseConnectionManager.instance.get_connection
    # Écrire les préférences
  end
end

class PrefReader
  def read
    connection = DatabaseConnectionManager.instance.get_connection
    # Lire et retourner les préférences...
  end
end
```

Une approche améliorée consiste à concentrer l'information sur l'implémentation de `DatabaseConnectionManager` dans la classe `PreferenceManager` et simplement passer l'instance aux objets `reader` et `writer` :

```
class PreferenceManager
  def initialize
    @reader = PrefReader.new
    @writer = PrefWriter.new
    @preferences = { :display_splash => false,
    ➥:background_color => :blue }
  end

  def save_preferences
    preferences = {}
    @writer.write(DatabaseConnectionManager.instance, @preferences)
  end
```

```
    def get_preferences
      @preferences = @reader.read(DatabaseConnectionManager.instance)
    end
  end
```

Cette refactorisation diminue le volume de code qui est au courant du statut spécial de `DatabaseConnectionManager`. Cela présente deux avantages. Premièrement, il y aurait moins de code à corriger, si vous découvrez que votre singleton n'est finalement pas si unique que cela. Deuxièmement, le fait d'enlever le singleton des classes `PrefReader` et `PrefWriter` permet de les tester beaucoup plus facilement.

Un remède contre les maux liés aux tests

Le dernier point concerne les tests. Un inconvénient très fâcheux du pattern Singleton tient à sa façon d'interférer avec des tests unitaires. Un bon test unitaire doit démarrer dans un état bien déterminé. En effet, le résultat de votre test ne serait pas très fiable si vous n'étiez pas certain des conditions initiales du test. Un bon test unitaire doit également être indépendant des autres tests : le test 3 est censé retourner exactement les mêmes résultats, peu importe s'il est exécuté entre les tests 2 et 4, après le test 20 ou tout seul. Mais, si les tests de 1 à 20 vérifient le fonctionnement d'un singleton, chacun d'eux est susceptible de modifier la seule instance du singleton de façon imprévisible. Dans ces conditions, l'indépendance de tests n'existe plus.

Une des solutions à ce problème consiste à créer deux classes : une classe ordinaire (non singleton) qui contient tout le code, et sa sous-classe qui est un singleton :

```
require 'singleton'
class SimpleLogger
  # Toute la fonctionnalité de logs est dans cette classe...
end
class SingletonLogger < SimpleLogger
  include Singleton
end
```

L'application fait appel à `SingletonLogger`, alors que les tests peuvent utiliser la classe normale `Logger`, qui n'est pas un singleton.

Les singletons dans le monde réel

Un bon exemple du pattern Singleton se trouve dans ActiveSupport, la bibliothèque des assistants fournis Rails. Rails s'appuie énormément sur le principe des conventions. Une des nombreuses conventions de Rails consiste à passer de la forme singulier d'un mot à la forme pluriel et inversement. Pour y parvenir, ActiveSupport maintient une liste de règles telles que "le pluriel du mot *employee* est *employees*, mais le pluriel du

mot *criterion* est *criteria*". Puisque ce sont des règles, seul un exemplaire de ces règles est nécessaire dans le système. Voilà pourquoi la classe `Inflections` est un singleton : cela économise l'espace et assure que les mêmes règles de pluralisation sont disponibles dans la totalité du système.

L'utilitaire d'installation rake utilise également un singleton. Tout comme les autres utilitaires d'installation, rake lit à l'exécution les informations sur les tâches à effectuer : quels dossiers il faut créer et quels fichiers copier, etc.[1] Toute l'information doit rester disponible à l'ensemble des composants de rake, c'est pourquoi rake stocke les données dans un objet unique (l'objet `Rake::Application` pour être précis) qui est accessible à la totalité du programme en tant que singleton.

En conclusion

Dans ce chapitre, nous avons passé en revue la carrière quelque peu chaotique du pattern Singleton. Ce pattern peut nous aider à gérer des cas où il n'existe qu'une occurrence d'un objet. Un singleton présente deux caractéristiques fondamentales : la classe singleton a exactement une instance et cette instance est accessible globalement. L'utilisation des méthodes et des variables de classe nous permet d'implémenter facilement un singleton classique recommandé par le GoF.

D'autres méthodes sont disponibles pour développer des singletons ou tout au moins des presque singletons. Par exemple, on pourrait atteindre en partie le comportement de singleton à l'aide des variables globales et des constantes. Néanmoins, ces éléments ne peuvent assurer la propriété indispensable d'unicité d'un singleton. Qui plus est, nous pouvons construire un singleton avec des méthodes et variables attachées à une classe ou à un module.

Dans ce chapitre, nous avons consacré beaucoup d'attention aux pièges tendus par les singletons. Ce pattern présente des dispositions particulières pour augmenter fortement le couplage de votre code. Nous avons appris qu'il est judicieux de limiter le volume de code qui est au courant du statut spécial d'un objet singleton. Nous avons également vu un moyen de lever les contraintes que ce pattern impose sur les tests unitaires.

Le pattern Singleton me rappelle une scie de mon père. Elle était incroyablement efficace dans le découpage du bois, mais faute de dispositifs de sécurité elle était tout aussi bien capable de vous trancher la main.

---

1. L'utilitaire rake applique le pattern Internal DSL (voir Chapitre 16) pour la plupart de la lecture.

13

Choisir la bonne classe avec Factory

Mon professeur de physique à l'école était un de ces enseignants extraordinaires capables de rendre vivant et intéressant le plus ennuyeux des sujets. Au bout de deux mois, tous les élèves dans les cours de physique élémentaire semblaient avoir abandonné leur unique ambition d'avoir une bonne note pour un but plus noble : "faire véritablement de la physique". Cela impliquait bien des choses comme faire les expériences avec beaucoup de soin, réfléchir énormément. Il y avait encore une chose qu'un bon élève de physique devait éviter à tout prix : l'attitude désinvolte. Les actions comme omettre un détail clé, bidouiller une équation ou supposer quelque chose non confirmé par l'expérience illustrent bien cette notion.

Je dois avouer qu'en écrivant ce livre je me suis rendu coupable d'une attitude désinvolte. J'ai en effet omis le mécanisme qui permet à votre code de savoir comme par magie quelle classe choisir à un moment critique. D'habitude, sélectionner la bonne classe n'est pas difficile : lorsqu'on a besoin d'un objet `String` ou `Date` ou même `PersonnelRecord`, on appelle simplement la méthode `new` sur la classe correspondante. Mais, parfois, le choix de la classe est une décision cruciale et se retrouver dans ce genre de situation est très courant. Pensez au pattern Template Method, par exemple. Lorsque vous utilisez le pattern Template Method, vous devez sélectionner une sous-classe, et ce choix détermine la variante d'algorithme qui sera exécutée. Est-ce `PlainReport` ou `HTMLReport` que vous voulez utiliser ? De la même manière, avec le pattern Strategy, il faut choisir la stratégie adaptée pour passer à votre objet de contexte : est-ce que vous avez besoin de `FrenchTaxCalculator` ou d'`ItalianTaxCalculator` ? C'est aussi valable pour le pattern Proxy : vous devez sélectionner la classe proxy avec le comportement nécessaire.

Il existe plusieurs façons de déterminer la classe la mieux adaptée aux circonstances, y compris deux patterns du GoF. Dans ce chapitre, nous examinerons les deux patterns : Factory Method et Abstract Factory. Nous apporterons quelques éclaircissements à certaines techniques dynamiques du langage Ruby qui peuvent nous aider à développer ces fabriques (*factories*) de façon plus efficace.

Une autre sorte de typage à la canard

Pour débuter notre exploration des factories, commençons par un problème de programmation. Imaginez qu'on vous demande de développer un simulateur de mare à canard. Plus précisément, on vous demande de créer un modèle permettant de simuler la vie d'un canard. Vous écrivez donc une classe pour modéliser un canard :

```
class Duck
  def initialize(name)
    @name = name
  end

  def eat
    puts("Duck #{@name} is eating.")
  end

  def speak
    puts("Duck #{@name} says Quack!")
  end

  def sleep
    puts("Duck #{@name} sleeps quietly.")
  end
end
```

On peut voir dans ce code que les canards mangent, dorment et font du bruit, tout comme les autres animaux. Ils ont aussi besoin d'un endroit pour vivre. Développons donc la classe `Pond` (NDT : la mare à canard en anglais) :

```
class Pond
  def initialize(number_ducks)
    @ducks = []
    number_ducks.times do |i|
      duck = Duck.new("Duck#{i}")
      @ducks << duck
    end
  end

  def simulate_one_day
    @ducks.each {|duck| duck.speak}
    @ducks.each {|duck| duck.eat}
    @ducks.each {|duck| duck.sleep}
  end
end
```

Exécuter le simulateur ne présente pas de difficultés :

```
pond = Pond.new(3)
pond.simulate_one_day
```

Le code précédent est une simulation d'une journée ordinaire dans la mare où vivent trois canards. Il affiche le résultat suivant :

```
Duck Duck0 says Quack!
Duck Duck1 says Quack!
Duck Duck2 says Quack!
Duck Duck0 is eating.
Duck Duck1 is eating.
Duck Duck2 is eating.
Duck Duck0 sleeps quietly.
Duck Duck1 sleeps quietly.
Duck Duck2 sleeps quietly.
```

L'idylle à l'étang continue jusqu'à ce que vous soyez amené à créer un modèle d'un autre habitant : la grenouille. Créer une classe `Frog (grenouille)` est facile car elle présente exactement la même interface que la classe `Duck` :

```
class Frog
  def initialize(name)
    @name = name
  end

  def eat
    puts("Frog #{@name} is eating.")
  end

  def speak
    puts("Frog #{@name} says Crooooaaaak!")
  end

  def sleep
    puts("Frog #{@name} doesn't sleep; he croaks all night!")
  end
end
```

Mais un problème surgit dans la classe `Pond`. On crée des canards explicitement dans sa méthode `initialize` :

```
def initialize(number_ducks)
  @ducks = []
  number_ducks.times do |i|
    duck = Duck.new(nDuck#{i}n)
    @ducks << duck
  end
end
```

Le problème, c'est que nous avons besoin de séparer la partie variable – les animaux qui habitent l'étang (des canards ou des grenouilles) – des parties statiques comme les

détails de fonctionnement de la classe `Pond`. S'il y avait un moyen d'extraire l'appel `Duck.new` de la classe `Pond`, cette classe pourrait supporter des canards ainsi que des grenouilles. Ce dilemme nous amène à la question principale de ce chapitre : quelle classe faut-il utiliser ?

Le retour du pattern Template Method

Une des façons de gérer le problème consiste à déléguer le choix de la classe à une sous-classe. On commence par développer une classe parent générique. Elle est générique dans le sens où elle ne sélectionne pas la classe à utiliser. Lorsque la classe parent a besoin d'un nouvel objet, elle appelle une méthode définie dans une sous-classe. Par exemple, on pourrait remanier la classe `Pond` afin qu'elle se fonde sur la méthode `new_animal` pour instancier les habitants de l'étang :

```
class Pond
  def initialize(number_animals)
    @animals = []
    number_animals.times do |i|
      animal = new_animal("Animal#{i}")
      @animals << animal
    end
  end

  def simulate_one_day
    @animals.each { | animal | animal.speak}
    @animals.each { | animal | animal.eat}
    @animals.each {|animal| animal.sleep}
  end
end
```

Ensuite, on peut définir deux sous-classes de `Pond`, une pour un étang plein de canards et une autre pour un étang envahi de grenouilles :

```
class DuckPond < Pond
  def new_animal(name)
    Duck.new(name)
  end
end

class FrogPond < Pond
  def new_animal(name)
    Frog.new(name)
  end
end
```

Désormais, il suffit de sélectionner le type de l'étang et il serait automatiquement rempli des habitants du type correspondant :

```
pond = FrogPond.new(3)
pond.simulate_one_day
```

On obtient des résultats qui sentent bon la vase et l'herbe verte :

```
Frog Animal10 says Crooooaaaak!
Frog Animal11 says Crooooaaaak!
Frog Animal12 says Crooooaaaak!
Frog Animal10 is eating.
Frog Animal11 is eating.
...
```

Je n'inclus pas cet exemple, mais on pourrait tout aussi bien créer une sous-classe de `Pond` dont la méthode `new_animal` produit un mélange de canards et de grenouilles.

Les membres du GoF ont nommé Factory Method le pattern qui consiste à déléguer le choix de la classe vers une sous-classe. Son diagramme de classe UML compte deux hiérarchies de classes (voir Figure 13.1). D'une part, il y a des créateurs : la classe parent et des classes concrètes qui contiennent les méthodes de création. D'autre part, nous avons des produits : des objets que l'on instancie. Dans notre exemple avec l'étang, la classe `Pond` est un créateur, alors que les types spécifiques d'étangs (`DuckPond` et `FrogPond`) sont des créateurs concrets ; les classes `Duck` et `Frog` sont des produits.

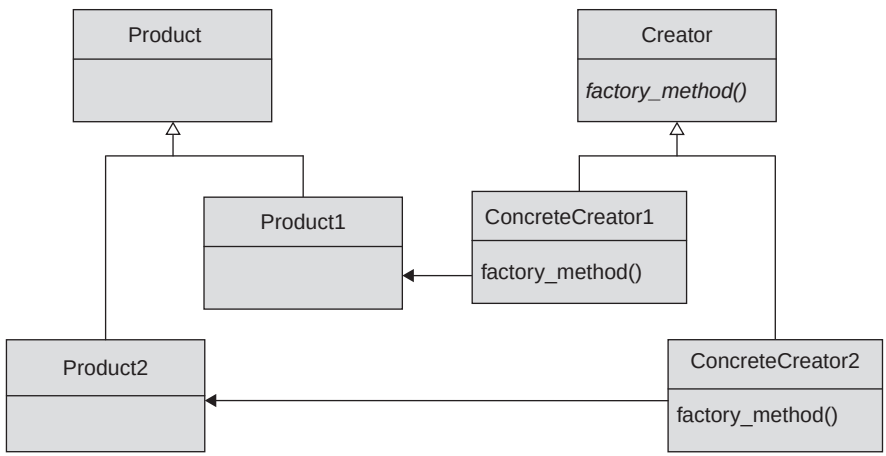


Figure 13.1
Diagramme de classe du pattern Factory Method

Malgré le fait qu'elles partagent la même classe parent `Product`, les classes `Duck` et `Frog` ne sont pas apparentées (voir Figure 13.1). Elles sont du même type car elles implémentent le même ensemble de méthodes.

Si vous scrutez la Figure 13.1 attentivement, vous découvrirez que le pattern Factory Method n'est pas un nouveau pattern. C'est simplement le pattern Template Method (souvenez-vous du Chapitre 3) appliqué au problème d'instanciation d'objets. Dans les

deux patterns, Factory Method et Template Method, une partie générique de l'algorithme (dans notre exemple, c'est l'activité journalière d'un étang) est codée dans la classe parent, et les sous-classes comblent les lacunes de cette classe. Dans le pattern Factory Method, les sous-classes déterminent les classes des objets qui peuplent l'étang.

Des méthodes factory avec des paramètres

Un des problèmes avec les programmes qui ont du succès est leur tendance à attirer sans cesse de nouvelles demandes de fonctionnalités. Imaginez que votre simulateur d'étang soit devenu populaire au point que vos utilisateurs vous demandent d'ajouter des modèles de plantes. Vous levez votre baguette magique et définissez quelques classes représentant des plantes :

```
class Algae
  def initialize(name)
    @name = name
  end

  def grow
    puts("The Algae #{@name} soaks up the sun and grows")
  end
end

class WaterLily
  def initialize(name)
    @name = name
  end

  def grow
    puts("The water lily #{@name} floats, soaks up the sun, and grows")
  end
end
```

Vous modifiez également la classe `Pond` pour prendre en compte les plantes :

```
class Pond
  def initialize(number_animals, number_plants)
    @animals = []
    number_animals.times do |i|
      animal = new_animal("Animal#{i}")
      @animals << animal
    end
    @plants = []
    number_plants.times do |i|
      plant = new_plant("Plant#{i}")
      @plants << plant
    end
end
```

```
  def simulate_one_day
    @plants.each {|plant| plant.grow }
    @animals.each {|animal| animal.speak}
    @animals.each {|animal| animal.eat}
    @animals.each {|animal| animal.sleep}
  end
end
```

Il faudrait apporter des modifications aux sous-classes pour créer la végétation :

```
class DuckWaterLilyPond < Pond
  def new_animal(name)
    Duck.new(name)
  end

  def new_plant(name)
    WaterLily.new(name)
  end
end

class FrogAlgaePond < Pond
  def new_animal(name)
    Frog.new(name)
  end

  def new_plant(name)
    Algae.new(name)
  end
end
```

Notre code est quelque peu maladroit car nous avons une méthode séparée pour chaque type d'objet produit : `new_animal` pour instancier des grenouilles et des canards et `new_plant` pour créer des nénuphars et des algues. Avoir des méthodes séparées pour chaque type d'objet n'est pas pénalisant s'il n'existe que deux types comme dans notre exemple. Et si vous aviez cinq ou dix types différents ? Coder toutes ces méthodes deviendrait vite très fastidieux.

Une solution alternative, et probablement plus propre, s'appuierait sur une méthode factory unique acceptant en paramètre le type d'objet qui doit être créé. Le code suivant présente la classe `Pond` une fois de plus. Cette fois, elle expose une méthode factory qui accepte des paramètres et qui est capable de produire une plante ou un animal en fonction du symbole passé en argument :

```
class Pond
  def initialize(number_animals, number plants)
    @animals = [] number_animals.times do |i|
      animal = new_organism(:animal, "Animal#{i}")
      @animals << animal
    end
```

```
    @plants = []
      number plants.times do |i|
        plant = new_organism(:plant, "Plant#{i}")
        @plants << plant
      end
    end
  # ...
  end

  class DuckWaterLilyPond < Pond
    def new_organism(type, name)
      if type == :animal
        Duck.new(name)
      elsif type == :plant
        WaterLily.new(name)
      else
        raise "Unknown organism type: #{type}"
      end
    end e
  end
```

Les méthodes factory à paramètres amincissent considérablement votre code car les sous-classes ne définissent qu'une seule méthode factory. Elles rendent également l'ensemble du système plus facilement extensible. Supposez qu'il faille définir un nouvel habitant pour votre étang : par exemple le poisson. Dans ce cas, il suffit de modifier une seule méthode des sous-classes au lieu d'ajouter une nouvelle méthode : c'est un autre exemple des avantages apportés par la séparation des parties variables et statiques.

Les classes sont aussi des objets

La nécessité de créer une nouvelle sous-classe pour chaque type d'objet à instancier constitue un sérieux argument contre notre implémentation actuelle du pattern Factory Method. Les noms des sous-classes dans notre dernière version reflètent cet état de fait : on a `DuckWaterLilyPond` et `FrogAlgaePond`, mais on pourrait aussi avoir besoin de `DuckAlgaePond` ou de `FrogWaterLilyPond`. Lorsque la collection de plantes et d'animaux s'étoffe, le nombre de sous-classes possibles devient effrayant. Mais, en réalité, la seule différence entre ces étangs divers et variés se réduit à la classe des objets instanciés par la méthode factory : ce sont parfois des nénuphars et des canards et parfois des algues et des grenouilles.

Il faut se rendre compte que les classes `Frog`, `Duck`, `WaterLily` et `Algae` sont des objets classiques qui passent leur vie à créer d'autres objets. En conservant les classes d'objets à créer dans des variables d'instance, on pourrait se débarrasser de la hiérarchie toute entière des sous-classes de `Pond` :

```
class Pond
  def initialize(number_animals, animal_class,
    number plants, plant_class)
    @animal_class = animal_class
    @plant_class = plant_class
    @animals = []
    number_animals.times do |i|
      animal = new_organism(:animal, "Animal# {i} " )
      @animals << animal
    end
    @plants = []
    number plants.times do |i|
      plant = new_organism(:plant, "Plant#{i}")
      @plants << plant
    end
  end

  def simulate_one_day
    @plants.each {|plant| plant.grow}
    @animals.each {|animal| animal.speak}
    @animals.each {|animal| animal.eat}
    @animals.each {|animal| animal.sleep}
  end

  def new_organism(type, name)
    if type == :animal
      @animal class.new(name)
    elsif type == :plant
      @plant_class.new(name)
    else
      raise "Unknown organism type: #{type}"
    end
  end
end
```

La nouvelle classe `Pond` n'est pas plus complexe d'utilisation que la version précédente. Nous passons les classes des plantes et des animaux dans le constructeur :

```
pond = Pond.new(3, Duck, 2, WaterLily)
pond.simulate_one_day
```

Le fait de stocker les classes d'animaux et de plantes dans `Pond` nous permet de réduire le nombre de classes nécessaire à une seule. Et, de plus, nous avons obtenu ce résultat sans rendre la classe `Pond plus complexe`.

Mauvaise nouvelle : votre programme a du succès

Supposons que votre simulateur d'étangs gagne encore davantage en popularité et que de nouvelles exigences tombent sur votre bureau plus vite que jamais. La demande la plus urgente est le besoin de gérer des environnements autres qu'un étang. En effet, un

commercial en grande forme vient de signer une commande pour un simulateur de jungle.

Il est évident que cette modification ne serait guère possible sans une intervention majeure dans le code. On aurait certainement besoin de modèles d'animaux de la jungle (des tigres, peut-être) ainsi que de plantes (principalement des arbres) :

```
class Tree
  def initialize(name)
    @name = name
  end

  def grow
    puts("The tree #{@name} grows tall")
  end
end

class Tiger
  def initialize(name)
    @name = name
  end

  def eat
    puts("Tiger #{@name} eats anything it wants.")
  end

  def speak
    puts("Tiger #{@name} Roars!")
  end

  def sleep
    puts("Tiger #{@name} sleeps anywhere it wants.")
  end
end
```

Il faudrait également transformer le nom de la classe `Pond` en un nom plus générique et adaptable à un étang aussi bien qu'à une jungle. `Habitat` semble approprié :

```
jungle = Habitat.new(1, Tiger, 4, Tree)
jungle.simulate_one_day
pond = Habitat.new( 2, Duck, 4, WaterLily)
pond.simulate_one_day
```

Mis à part le changement de nom, notre classe `Habitat` est identique à la dernière version de la classe `Pond` (celle qui avait des classes de plantes et d'animaux). Des habitats se créent de la même façon que des étangs.

Création de lots d'objets

L'un des problèmes de notre nouvelle classe `Habitat` tient à la possibilité de créer des combinaisons de flore et de faune incohérentes d'un point de vue écologique. Par

exemple, aucun mécanisme n'est prévu dans notre implémentation actuelle pour nous prévenir que des tigres ne sont pas compatibles avec des nénuphars :

```
unstable = Habitat.new( 2, Tiger, 4, WaterLily)
```

Cela ne représente pas un problème majeur lorsque nous n'avons que deux types de choses à gérer (des plantes et des animaux dans ce cas). Mais il n'en irait pas de même si la simulation était plus détaillée et qu'elle incorporait des insectes ainsi que des oiseaux, des mollusques et des champignons ? On voudrait certainement éviter des champignons qui poussent sur des nénuphars et des bancs de poissons qui se débattent dans les branches d'un arbre tropical.

Nous pouvons aborder ce problème en indiquant à quel habitat appartiennent des organismes particuliers. Au lieu de passer à `Habitat` des classes individuelles de plantes et d'animaux, on pourrait lui passer un objet unique qui maîtrise la création d'une collection cohérente de produits. Une version de cet objet existerait pour des étangs : il créerait des instances de grenouilles et nénuphars. Une deuxième version serait propre à l'environnement de la jungle et instancierait des tigres ainsi que des arbres tropicaux. Un objet dédié à la création d'un certain nombre d'objets compatibles s'appelle une factory abstraite ou Abstract Factory, un pattern que le GoF a rendu célèbre. Le code ci-après présente deux factories abstraites de notre simulateur d'habitats. La première correspond à un étang et la deuxième, à une jungle :

```
class PondOrganismFactory
  def new_animal(name)
    Frog.new(name)
  end

  def new_plant(name)
    Algae.new(name)
  end
end

class JungleOrganismFactory
  def new_animal(name)
    Tiger.new(name)
  end

  def new plant(name)
    Tree.new(name)
  end
end
```

Après quelques modifications simples, la méthode `initialize` de notre classe `Habitat` est prête à utiliser la factory abstraite :

```
class Habitat
  def initialize(number_animals, number_plants, organism_factory)
```

```
      @organism_factory = organism_factory
      @animals = []
      number_animals.times do |i|
        animal = @organism_factory.new_animal("Animal#{i}")
        @animals << animal
      end
      @plants = []
      number_plants.times do |i|
        plant = @organism_factory.new_plant("Plant#{i}")
        @plants << plant
      end
    end
  # Le reste de la classe...
```

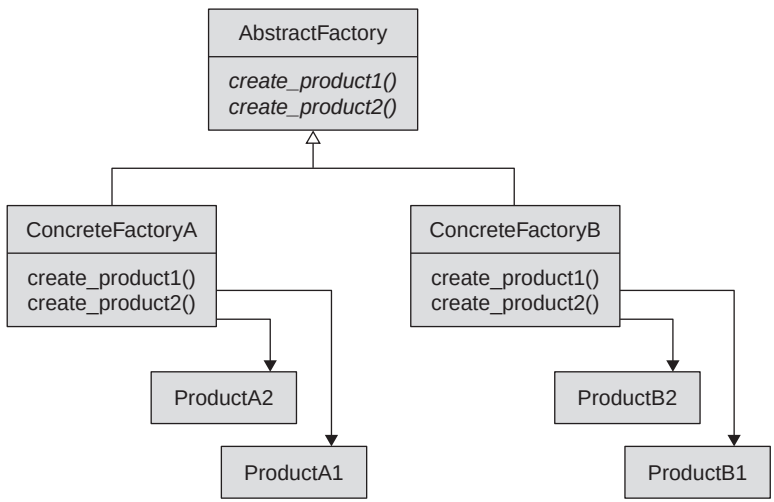
Nous pouvons désormais fournir la factory abstraite à notre `Habitat` en étant sûr de ne pas nous retrouver avec un étrange mélange d'habitants des étangs et de la jungle :

```
jungle = Habitat.new(1, 4, JungleOrganismFactory.new)
jungle.simulate_one_day
pond = Habitat.new(2, 4, PondOrganismFactory.new)
pond.simulate_one_day
```

La Figure 13.2 présente le diagramme des classes UML du pattern Abstract Factory. On voit deux classes factory concrètes qui créent leurs groupes respectifs de produits compatibles.

Figure 13.2

Le pattern Abstract Factory



L'essentiel du pattern Abstract Factory réside dans l'expression du problème et de sa solution. Le problème est ici le besoin de créer un ensemble d'objets compatibles. La solution consiste à écrire une classe séparée pour gérer l'instanciation. Le pattern

Factory Method est en réalité le pattern Template Method appliqué à la création d'objets. De la même manière, le pattern Abstract Factory n'est autre que le pattern Strategy appliqué au même problème.

Des classes sont des objets (encore)

On peut voir la factory abstraite comme un objet d'un niveau supérieur : elle est capable de créer plusieurs types d'objets (ou produits) alors que les classes normales ne savent créer qu'un seul type d'objet (des instances d'elles-mêmes). Cette vision nous suggère un moyen de simplifier notre implémentation du pattern Abstract Factory : elle peut se fonder sur des objets classe, une classe par produit. C'est exactement la même approche adoptée plus haut pour simplifier le pattern Factory Method.

Le code ci-après démontre une factory abstraite fondée sur des classes. Au lieu d'avoir plusieurs factories abstraites dont chacune crée son propre groupe de produits, on peut définir une classe qui stocke les objets classe des produits à instancier :

```
class OrganismFactory
  def initialize(plant_class, animal_class)
    @plant_class = plant_class
    @animal_class = animal_class
  end

  def new_animal(name)
    @animal_class.new(name)
  end

  def new plant(name)
    @plant_class.new(name)
  end
end
```

Cette approche nous permet de créer une nouvelle instance de factory pour chaque groupe d'objets compatibles :

```
jungle_organism_factory = OrganismFactory.new(Tree, Tiger)
pond_organism_factory = OrganismFactory.new(WaterLily, Frog)
jungle = Habitat.new(1, 4, jungle_organism_factory)
jungle.simulate_one_day
pond = Habitat.new( 2, 4, pond_organism_factory)
pond.simulate_one_day
```

Cela peut ressembler à un serpent qui se mord la queue : nous avons commencé par définir une factory abstraite pour éviter de spécifier des classes individuelles et il semble que notre dernière implémentation laisse la porte ouverte à la création d'étangs pleins de tigres ou d'une jungle envahie par des algues. Ce n'est pas le cas car une factory abstraite permet l'encapsulation de l'information concernant la compatibilité

des types de produits. Cette encapsulation peut être exprimée par des classes et sous-classes ainsi que par des objets classe comme dans le code ci-dessus. Dans tous les cas, vous vous retrouvez avec un objet qui sait quelles choses sont liées ou interdépendantes.

Profiter du nommage

Un autre moyen de simplifier l'implémentation d'une factory abstraite consiste à adopter une convention de nommage cohérente pour les classes de produits. Cette approche n'est pas applicable dans notre exemple à la classe `Habitat`, car cette classe contient des tigres et des grenouilles qui ont chacun un nom unique. Mais imaginez que vous deviez produire une factory abstraite pour des objets capables de lire et d'écrire différents formats de fichiers tels que PDF, HTML et PostScript. On pourrait certainement implémenter une classe `IOFactory` à l'aide d'une des techniques exposées dans ce chapitre. Mais, si les noms des classes de lecture et d'écriture suivent une règle prédéfinie, quelque chose comme `HTMLReader` / `HTMLWriter` pour du HTML et `PDFReader` / `PDFWriter` pour le format PDF, on pourrait déduire le nom de la classe à partir du nom du format. Le code ci-dessous met cette démarche en œuvre :

```
class IOFactory
  def initialize(format)
    @reader_class = self.class.const_get("#{format}Reader")
    @writer_class = self.class.const_get("#{format}Writer")
  end

  def new_reader
    @reader_class.new
  end

  def new_writer
    @writer_class.new
  end
end
html_factory = IOFactory.new('HTML')
html_reader = html_factory.new_reader
pdf_factory = IOFactory.new('PDF')
pdf_writer = pdf_factory.new_writer
```

La méthode `const_get` de la classe `IOFactory` accepte une chaîne de caractères (ou un symbole) qui contient le nom d'une constante[1] et retourne sa valeur. Par exemple, si vous passez dans la méthode `const_get` la chaîne `"PDFWriter"`, vous obtiendrez en retour un objet classe avec ce nom : exactement ce que l'on souhaite[2].

---

1. Souvenez-vous que les noms de classes en Ruby sont des constantes.
2. Évidemment, si la classe `PDFWriter` n'existe pas, la méthode `const_get` déclencherait une exception. Dans notre cas, c'est aussi le comportement souhaité.

User et abuser des patterns Factory

Le moyen le plus simple de se tromper dans l'utilisation des techniques de création d'objets décrites dans ce chapitre est d'y recourir sans en avoir besoin. Il ne faut pas instancier tous les objets avec des factories. En effet, dans la plupart des cas il serait judicieux de créer vos objets avec un simple appel `MyClass.new`. Servez-vous des factories lorsque vous devez faire un choix parmi des classes distinctes mais ayant un lien entre elles.

N'oubliez pas le principe YAGNI (*You Ain't Gonna Need It*), qui dit que dans l'immense majorité des cas "vous n'aurez pas besoin de ça". Ce principe s'applique sans doute aux factories plus qu'à tous autres patterns. Pour le moment, je ne gère que des canards et des nénuphars. Il est possible que dans le futur j'aie besoin de traiter des tigres et des arbres. Devrais-je développer une factory pour m'y préparer ? Probablement non. Il faut trouver l'équilibre entre le coût engendré par une factory supplémentaire, sûrement inutile au début, et la probabilité d'en avoir effectivement besoin ultérieurement. N'oubliez pas d'inclure le coût des ajustements futurs de la factory. La réponse dépend des détails mais, en règle générale, les ingénieurs ont tendance à construire un cargo là où un canoë est souvent suffisant. Si pour le moment vous n'avez qu'une classe à sélectionner, remettez à plus tard l'implémentation d'une fabrique.

Les factories dans le monde réel

Il s'avère que les implémentations classiques des factories fondées sur l'héritage sont assez difficiles à trouver dans la base de code Ruby. Les programmeurs Ruby semblent avoir voté pour des versions de factories plus dynamiques, qui se fondent sur des objets classe ou sur différentes conventions de nommage des classes. Ainsi, la bibliothèque SOAP distribuée avec l'interpréteur Ruby instancie des objets Ruby en fonction des chaînes de caractères XML correspondant à des noms des classes. De la même manière, l'implémentation de XMLRPC[1] incluse dans la bibliothèque standard Ruby supporte plusieurs options d'analyse syntaxique de XML. Chacune de ces méthodes d'analyse de XML a une classe d'analyseur syntaxique associée. Il existe une classe pour convertir un fichier XML en un flux et une autre pour le transformer en un arbre DOM. Mais il n'y a pas de sous-classes qui correspondent à chaque technique de traitement. Le code XMLRPC conserve la classe de l'analyseur XML sélectionné et produit de nouvelles instances d'analyseurs à partir de l'objet classe si besoin.

---

1. Si vous ne l'avez pas encore rencontré, XMLRPC est un mécanisme d'appel de procédures distantes fondé sur du XML, semblable à SOAP. Contrairement à SOAP, XMLRPC fait son possible pour simplifier le protocole des communications.

Une version relativement exotique du pattern Factory Method se trouve dans ActiveRecord. Nous avons appris au Chapitre 9 qu'ActiveRecord possède une classe adaptateur pour chaque type de base de données supporté. C'est ainsi qu'on trouve des adaptateurs pour MySQL, Oracle, DB2, etc. Lorsque vous demandez à ActiveRecord d'établir une connexion avec la base de données, à part le nom d'utilisateur, le mot de passe et le port vous devez spécifier une chaîne avec le nom de l'adaptateur à utiliser. Vous passez "mysql" si vous voulez qu'ActiveRecord se connecte sur une base MySQL ou "oracle" si c'est une base Oracle. Mais comment ActiveRecord arrive-t-il à récupérer une instance d'adaptateur à partir de cette chaîne de caractères ?

Il s'avère qu'ActiveRecord fait appel à une technique assez intéressante pour retrouver les instances d'adaptateurs. Le cœur de cette technique s'appuie sur la classe `Base`[1]. Cette dernière est complètement indépendante de tout adaptateur spécifique :

```
class Base
# Beaucoup de code omis. Ce code n'est pas lié aux adaptateurs...
end
```

Toutefois, chaque adaptateur contient du code apportant des modifications spécifiques à la classe `Base`. En d'autres termes, chaque adaptateur ajoute à la classe `Base` une méthode qui crée son propre type de connexion. Par exemple, l'adaptateur MySQL contient du code qui ressemble à ceci :

```
class Base
  def self.mysql_connection(config)
    # Créer et retourner une nouvelle connexion MySQL en utilisant
    # le nom d'utilisateur, le mot de passe, etc. qui sont définies
    # dans le tableau associatif de configuration...
  end
end
```

De la même manière, l'adaptateur Oracle contient le code suivant :

```
class Base
  def self.oracle_connection(config)
    # Créer une nouvelle connexion Oracle ...
  end
end
```

1. En réalité, la classe `Base` est définie hors du module ActiveRecord. Pour cette raison, son nom est habituellement écrit comme `ActiveRecord::Base`. Dans un intérêt de simplicité, j'ai supprimé de notre code le nom du module, ainsi que de nombreux autres détails qui ne sont pas pertinents dans cette section.

Une fois les modifications effectuées pour un adaptateur, la classe `Base` se retrouve avec une méthode appelée `<<db_type>>_connection` pour chaque type de base de données qu'elle est capable de gérer.

Pour créer une vraie connexion à partir d'un nom d'adaptateur, la classe `Base` construit une chaîne qui contient le nom de la méthode spécifique à la base de données. Le fonctionnement ressemble à ceci :

```
adapter = "mysql"
method_name = "#{adapter}_connection"
Base.send(method_name, config)
```

La dernière ligne de cette méthode appelle la méthode spécifique à la base de données et passe en paramètre la configuration de la connexion : par exemple, le nom de la base, le nom d'utilisateur et le mot de passe. La connexion est établie !

En conclusion

Dans ce chapitre, nous avons étudié deux patterns Factory du GoF. Les deux techniques répondent à la question "Quelle classe choisir ?".

Le pattern Factory Method n'est rien d'autre que l'application du pattern Template Method à la création d'objets. Fidèle à ses racines Template Method, ce pattern laisse le soin à une sous-classe de sélectionner la bonne classe pour instancier l'objet qui nous intéresse. Nous avons réussi à appliquer ce pattern pour développer la classe `Pond` générique qui encapsule les détails de la simulation environnementale mais délègue le choix de plantes et d'animaux à sa sous-classe. Nous pouvons donc créer des sous-classes nommées `DuckWaterLilyPond` ou `FrogAlgaePond` qui complètent les implémentations des méthodes factory afin de créer des objets appropriés.

Le pattern Abstract Factory entre en jeu lorsque nous devons créer des groupes d'objets compatibles. Si vous souhaitez éviter que des grenouilles et des algues soient mélangées avec des tigres et des arbres, définissez une factory abstraite pour chaque combinaison valide.

Une des leçons à retenir de ce chapitre concerne la transformation que ces deux patterns ont subie une fois transposés dans l'environnement Ruby : ils sont devenus bien plus simples. Alors que le GoF se concentre sur des implémentations de factories fondées sur l'héritage, nous arrivons à obtenir le même résultat avec beaucoup moins de code. On profite du fait que les classes Ruby sont des objets : on peut les rechercher par leur nom, les passer en paramètre et les stocker pour les réutiliser plus tard.

Le pattern que nous allons examiner au chapitre suivant se nomme Builder. Il produit également de nouveaux objets, mais il se concentre davantage sur la construction des objets complexes que sur le choix de la bonne classe. Mais la question du choix des objets adaptés à une tâche donnée est loin d'être close. Au Chapitre 17, nous découvrirons la méta-programmation : une technique pour personnaliser vos classes et vos objets au moment de leur exécution.

14

Simplifier la création d'objets avec Builder

Je me rappelle très distinctement le jour où j'ai offert à mon fils son premier vélo. La matinée s'était bien passée : on avait pris la voiture pour se rendre au magasin, trouvé la bonne taille de vélo et passé beaucoup de temps sur le choix cornélien de la couleur. La phase intermédiaire consistait à rapporter le vélo à la maison et à sortir toutes les pièces de la boîte car, bien sûr, il fallait réaliser soi-même une partie non négligeable de l'assemblage. La troisième phase après notre heureux retour à la maison a provoqué une frustration profonde et quelques blessures aux doigts. J'ai passé des heures à essayer d'assembler la multitude de pièces selon des instructions qui auraient pu déconcerter une équipe entière de cryptologues. Choisir le vélo était facile, le vrai défi consistait à le monter.

De telles situations surviennent également avec les objets. Le Chapitre 13 a décrit des factories qui permettent de récupérer le bon type d'objet. Mais, parfois, récupérer l'objet n'est pas le problème principal. Parfois, le vrai souci est sa configuration.

Dans ce chapitre, nous examinerons le pattern Builder, qui est conçu pour vous aider à paramétrer des objets complexes. Nous verrons que le pattern Builder a un certain nombre de points communs avec le pattern Factory, ce qui n'a rien d'étonnant. Nous découvrirons également des méthodes magiques : une technique Ruby pour faciliter l'utilisation des objets builder. La question de la réutilisation de builders est un autre point évoqué dans ce chapitre. Enfin, nous allons apprendre la façon dont le pattern Builder peut vous aider à éviter des erreurs de paramétrage d'objets et même vous aider à créer des objets valides.

Construire des ordinateurs

Imaginez que vous développiez un système informatique pour une petite entreprise qui fabrique des ordinateurs. Chaque machine commandée est faite sur mesure, il sera donc nécessaire de garder trace des composants installés sur chacune des machines. Pour faire simple, supposons qu'un ordinateur se compose d'un écran, d'une carte mère et de plusieurs lecteurs de média amovibles :

```
class Computer
  attr_accessor :display
  attr_accessor :motherboard
  attr_reader :drives
  def initialize(display=:crt, motherboard=Motherboard.new, drives=[])
    @motherboard = motherboard
    @drives = drives
    @display = display
  end
end
```

Sélectionner un écran est simple, c'est soit `:crt` soit `:lcd`. L'objet `Motherboard` est plus complexe : il possède un certain volume de mémoire et contient soit un processeur normal, soit un processeur turbo très rapide :

```
class CPU
  # Caractéristiques générales d'un processeur...
end

class BasicCPU < CPU
  # Caractéristiques d'un processeur pas très rapide...
end

class TurboCPU < CPU
  # Caractéristiques d'un processeur très rapide...
end

class Motherboard
  attr_accessor :cpu
  attr_accessor :memory_size
  def initialize(cpu=BasicCPU.new, memory_size=1000)
    @cpu = cpu
    @memory_size = memory_size
  end
end
```

La classe Drive représente un lecteur qui est disponible en trois versions : disque dur, CD ou DVD :

```
class Drive
  attr_reader :type # peut être :hard_disk, :cd ou :dvd
  attr_reader :size # en MO
  attr_reader :writable # true si le lecteur est disponible en écriture
```

```
    def initialize(type, size, writable)
      @type = type
      @size = size
      @writable = writable
    end
  end
```

Même ce modèle simplifié engendre des difficultés lors de l'instanciation d'un objet `Computer` :

```
# Monter un ordinateur puissant avec beaucoup de mémoire...
motherboard = Motherboard.new(TurboCPU.new, 4000)
# ...avec un disque dur, un graveur CD et un lecteur DVD
drives= [ ]
drives<< Drive.new(:hard_drive, 200000, true)
drives<< Drive.new(:cd, 760, true)
drives<< Drive.new(:dvd, 4700, false)
computer = Computer.new(:lcd, motherboard, drives)
```

Le principe simple du pattern Builder consiste à encapsuler la logique de construction dans sa propre classe. La classe builder prend en charge l'assemblage de tous les composants d'un objet complexe. Chaque builder expose une interface permettant de spécifier étape par étape la configuration de votre nouvel objet. On peut voir l'objet builder comme une méthode `new` à plusieurs étapes. Les objets subissent un processus de construction étendu au lieu d'être créés instantanément. Le builder de nos ordinateurs pourrait ressembler à ceci :

```
class ComputerBuilder
  attr_reader :computer
  def initialize
    @computer = Computer.new
  end

  def turbo(has_turbo_cpu=true)
    @computer.motherboard.cpu = TurboCPU.new
  end

  def display=(display)
    @computer.display=display
  end

  def memory_size=(size_in_mb)
    @computer.motherboard.memory_size = size_in_mb
  end

  def add_cd(writer=false)
    @computer.drives << Drive.new(:cd, 760, writer)
  end

  def add_dvd(writer=false)
    @computer.drives << Drive.new(:dvd, 4000, writer)
  end
```

```
    def add_hard_disk(size_in_mb)
      @computer.drives << Drive.new(:hard_disk, size_in_mb, true)
    end
  end
```

La classe `ComputerBuilder` factorise tous les détails de la création d'un objet `Computer`. Il suffit de produire une nouvelle instance de builder et de spécifier pas à pas toutes les options requises pour votre ordinateur :

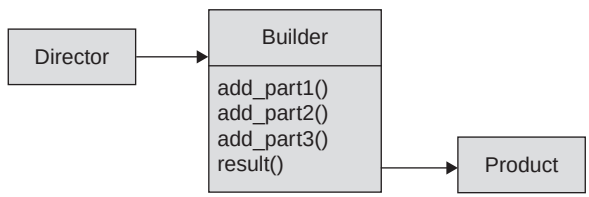
```
builder = ComputerBuilder.new
builder.turbo
builder.add_cd(true)
builder.add_dvd
builder.add_hard_disk(100000)
```

Enfin, vous obtenez une instance de votre `Computer` flambant neuf :

```
computer = builder.computer
```

La Figure 14.1 présente le diagramme de classes UML d'un objet builder basique.

Figure 14.1
Le pattern Builder



Les membres du GoF ont nommé le client de l'objet builder *director* car il guide le builder dans la construction d'un nouvel objet *produit*. Non seulement les builders facilitent la création des objets complexes, mais ils cachent également les détails de l'implémentation. L'objet director n'est pas obligé de maîtriser tous les points spécifiques de la création d'un nouvel objet. Lorsque l'on utilise la classe `ComputerBuilder`, on peut ignorer complètement quelle classe représente un lecteur DVD ou un disque dur : on souhaite simplement obtenir un ordinateur correspondant à la configuration requise.

Des objets builder polymorphes

Au début de ce chapitre, nous avons souligné la différence entre le pattern Builder et le pattern Factory : les builders sont moins concernés par le choix de la classe et se concentrent plus sur la configuration d'objets. Factoriser le code complexe lié à la configuration d'objets est la motivation principale des builders. Toutefois, puisqu'un builder gère la construction des objets, c'est un endroit commode pour prendre une décision sur le choix de la classe.

Par exemple, imaginez que notre entreprise informatique grandisse et commence à produire des ordinateurs portables en plus des ordinateurs de bureau. Nous avons désormais deux types de produits : des ordinateurs de bureau et des portables.

```
class DesktopComputer < Computer
  # Beaucoup de détails liés aux ordinateurs de bureau omis...
end

class LaptopComputer < Computer
  def initialize) motherboard=Motherboard.new, drives=[] )
    super(:lcd, motherboard, drives)
  end
  # Beaucoup de détails liés aux ordinateurs portables omis...
end
```

Il est évident que les composants d'un ordinateur portable ne sont pas les mêmes que ceux installés sur des ordinateurs de bureau. Heureusement que nous pouvons refactoriser la classe builder en une classe parent et deux sous-classes pour gérer ces différences. La classe abstraite prend en charge tous les détails communs à deux types d'ordinateurs :

```
class ComputerBuilder
  attr_reader :computer
  def turbo(has_turbo_cpu=true)
    @computer.motherboard.cpu = TurboCPU.new
  end

  def memory_size=(size_in_mb)
    @computer.motherboard.memory_size = size_in_mb
  end
end
```

La classe `DesktopBuilder` encapsule l'information sur la construction des ordinateurs de bureau. Plus particulièrement, elle est capable de créer des instances de la classe `DesktopComputer` et elle est au courant que l'on installe des lecteurs de média classiques sur des ordinateurs de bureau :

```
class DesktopBuilder < ComputerBuilder
  def initialize
    @computer = DesktopComputer.new
  end

  def display=(display)
    @display = display
  end

  def add_cd(writer=false)
    @computer.drives << Drive.new(:cd, 760, writer)
  end

  def add_dvd(writer=false)
    @computer.drives << Drive.new(:dvd, 4000, writer)
  end
```

```
    def add_hard_disk(size_in_mb)
      @computer.drives << Drive.new(:hard disk, size_in_mb, true)
    end
  end
```

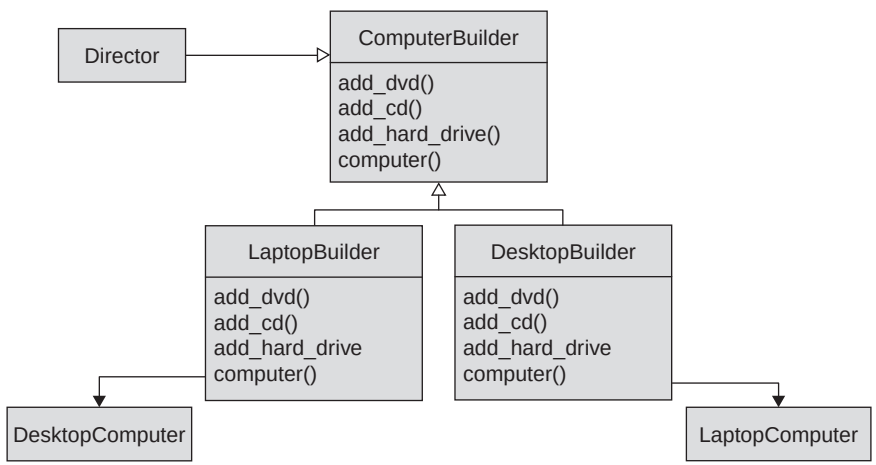
De son côté, la classe `LaptopBuilder` crée des objets `LaptopComputer` et leur affecte des instances d'un lecteur spécial `LaptopDrive` :

```
class LaptopBuilder < ComputerBuilder
  def initialize
    @computer = LaptopComputer.new
  end

  def display=(display)
    raise "Laptop display must be lcd" unless display == :lcd
  end

  def add_cd(writer=false)
    @computer.drives << LaptopDrive.new(:cd, 760, writer)
  end

  def add_dvd(writer=false)
    @computer.drives << LaptopDrive.new(:dvd, 4000, writer)
  end

  def add_hard_disk(size_in_mb)
    @computer.drives << LaptopDrive.new(:hard disk, size_in_mb, true)
  end
end
```

La Figure 14.2 présente le diagramme de classes UML de notre nouveau builder polymorphe. Si l'on compare la Figure 14.2 avec le diagramme de classes UML du pattern Abstract Factory (voir Figure 13.2), on remarque que ces patterns ont un certain air de famille.

Figure 14.2

L'implémentation d'un builder polymorphe



On pourrait également définir une classe builder unique qui crée un ordinateur portable ou un ordinateur de bureau en fonction de la valeur passée en argument.

Les builders peuvent garantir la validité des objets

Les objets builders rendent la création d'objets plus simple, mais ils peuvent aussi la rendre plus sûre. La méthode finale "rends-moi mon objet" est un endroit parfait pour vérifier que la configuration requise par le client est cohérente et compatible avec des règles de gestion appropriées. On pourrait par exemple améliorer notre méthode `computer` pour vérifier que la configuration est raisonnable :

```
def computer
  raise "Not enough memory" if @computer.motherboard.memory_size < 250
  raise "Too many drives" if @computer.drives.size > 4
  hard_disk = @computer.drives.find {|drive| drive.type == :hard_disk}
  raise "No hard disk." unless hard_disk
  @computer
end
```

Si la configuration est incomplète, nous avons la possibilité d'effectuer une correction au lieu de simplement déclencher une exception :

```
# ...
if ! hard_disk
  raise "No room to add hard disk." if @computer.drives.size >= 4
  add_hard_disk(100000)
end
# ...
```

Le code précédent ajoute un disque dur si le client ne l'a pas précisé et si l'emplacement pour le disque est disponible.

Réutilisation de builders

Lorsque vous écrivez et utilisez des builders, il faut vous poser une question importante : est-ce que votre instance de builder est capable de créer des objets multiples ? Par exemple, il est tout à fait possible de demander à votre `LaptopBuilder` deux instances d'ordinateurs identiques en une opération :

```
builder = LaptopBuilder.new
builder.add_hard_disk(100000)
builder.turbo
computer1 = builder.computer
computer2 = builder.computer
```

Étant donné que la méthode `computer` retourne toujours la même variable, `computer1` et `computer2` finissent par être des références vers le même ordinateur. Ce n'est probablement pas le comportement attendu. Une des solutions à cette difficulté consiste à équiper votre builder de la méthode `reset`, responsable de la réinitialisation de l'objet en construction :

```
class LaptopBuilder
  # Beaucoup de code omis...
  def reset
    @computer = LaptopComputer.new
  end
end
```

La méthode `reset` permet de réutiliser l'instance de builder, mais elle vous oblige également à recommencer la configuration pour chaque ordinateur. Si vous voulez paramétrer l'objet une seule fois et demander à l'objet builder de créer un nombre arbitraire d'objets suivant cette configuration, il faut conserver l'information sur la configuration dans des attributs d'instance et ne créer des produits réels que lorsqu'un client les demande.

Améliorer des objets builder avec des méthodes magiques

Employer notre builder d'ordinateurs est certainement une solution plus propre que répandre dans l'application le code qui gère la création, la configuration et la validation de nos objets. Malheureusement, même avec un builder, la procédure de configuration d'un ordinateur n'est pas très élégante. Nous avons vu que pour configurer chaque nouvel ordinateur il faut instancier le builder et appeler un certain nombre de ses méthodes. Comment pourrait-on rendre le processus de configuration d'un nouvel ordinateur plus concis et peut-être légèrement plus élégant ?

Une des façons d'y parvenir repose sur la création d'une méthode magique. L'idée sous-jacente à cette méthode magique laisse le client définir un nom de méthode selon un patron spécifique. On pourrait par exemple configurer un nouveau portable avec

```
builder.add_dvd_and_harddisk
```

ou encore

```
builder.add_turbo_and_dvd_and_harddisk
```

L'implémentation des méthodes magique est très simple si l'on recourt à la technique `method_missing` (nous l'avons déjà rencontrée au Chapitre 10, lorsque nous faisions connaissance avec les proxies). Pour utiliser la méthode magique, il suffit d'attraper

tous les appels inattendus à l'aide de `method_missing` et de vérifier que les noms de méthodes appelées correspondent aux patrons des noms de vos méthodes magiques :

```
def method_missing(name, *args)
  words = name.to_s.split("_")
  return super(name, *args) unless words.shift == 'add'
  words.each do |word|
    next if word == 'and'
    add_cd if word == 'cd'
    add_dvd if word == 'dvd'
    add_hard_disk(100000) if word == 'harddisk'
    turbo if word == 'turbo'
  end
end
```

Le code ci-dessus divise le nom de méthode en supprimant les tirets bas et tâche de traduire les fragments obtenus en requêtes sur les différentes options d'un ordinateur.

La technique de la méthode magique ne se limite pas au pattern Builder. Elle est applicable dans toute situation où vous souhaitez que le code client puisse spécifier des options multiples de façon concise.

User et abuser du pattern Builder

Le besoin pour le pattern Builder se fait souvent sentir lorsque votre application évolue vers un niveau de complexité élevé. Par exemple, supposons que tout au début votre classe `Computer` ne gère que les types de processeurs et le volume de la mémoire. L'utilisation d'un builder ne serait pas justifiée dans ces conditions. Lorsque vous améliorez la classe `Computer` pour prendre en compte des lecteurs de média, le nombre d'options et leurs interdépendances augmentent nettement. Le besoin pour un builder devient plus prononcé. D'habitude, il est relativement simple d'identifier le code où un builder est nécessaire : la logique de la création d'un objet particulier est dispersée partout dans le programme. Le fait que votre code commence à produire des objets invalides et que vous vous disiez à vous-même "J'ai vérifié le nombre de lecteurs lorsque j'ai créé un nouvel objet `Computer` à un endroit, mais j'ai oublié la vérification à un autre endroit dans le code" est un deuxième indicateur de la nécessité de mettre en place un builder.

Tout comme dans le cas du pattern Factory, l'erreur principale que vous pouvez être amené à faire consiste à appliquer le pattern Builder lorsque ce n'est pas nécessaire. Anticiper le besoin pour un builder n'est pas une bonne idée à mon avis. Par défaut, optez pour `MyClass.new` lorsque vous créez des objets. N'ajoutez un builder que lorsqu'une liste interminable de nouvelles demandes vous oblige à y recourir.

Des objets builder dans le monde réel

L'un des builders les plus intéressants que l'on trouve dans la base de code Ruby prétend ne pas en être un. Malgré le nom, `MailFactory`[1] est un builder sympathique qui vous aide à créer des messages électroniques. Alors que, fondamentalement, les messages électroniques ne sont que des fragments de texte simple, tout développeur qui a déjà tenté de construire des messages avec des pièces jointes en format MIME multipart sait que même le message le plus simple peut être très compliqué à créer.

`MailFactory` masque cette complexité à l'aide d'une agréable interface à la builder pour créer votre message :

```
require 'rubygems'
require 'mailfactory'
mail_builder = MailFactory.new
mail_builder.to ='russ@russolsen.com'
mail_builder.from = 'russ@russolsen.com'
mail_builder.subject = 'The document'
mail_builder.text = 'Here is that document you wanted'
mail_builder.attach('book.doc')
```

Une fois que vous avez informé `MailFactory` (le builder !) des détails de votre message, vous pouvez récupérer le texte du message dans un format approprié pour un serveur SMTP à l'aide de la méthode `to_s` :

```
puts mail_builder.to_s
to: russ@russolsen.com
from: russ@russolsen.com
subject: Here is that document you wanted
Date: Wed, 16 May 2007 14:02:32 -0400
MIME-Version: 1.0
Content-Type: multipart/mixed;
boundary="--=_NextPart_3rj.Kbd9.t9JpHIc663P_4mq6"
Message-ID: <1179338750.3053.1000.-606657668@russolsen.com>
This is a multi-part message in MIME format.
...
```

Les méthodes `find` d'ActiveRecord constituent un des exemples les plus remarquables de méthodes magiques. ActiveRecord nous permet d'encoder des requêtes vers la base de données avec exactement la même technique que celle employée dans la dernière version de notre builder pour spécifier les configurations à l'aide du nommage des méthodes. On pourrait par exemple retrouver tous les salariés de la table `Employee` par leur numéro de sécurité sociale :

```
Employee.find_by_ssn('123-45-6789')
```

1. L'auteur du package `MailFactory` est David Powers.

On peut également rechercher par le nom et le prénom :

```
Employee.find_by_firstname_and_lastname('John', 'Smith')
```

En conclusion

Lorsque vos objets sont difficiles à construire et qu'il faut écrire beaucoup de code pour les configurer, factorisez le code qui gère la création dans sa propre classe : telle est l'idée principale du pattern Builder.

Le pattern Builder stipule qu'il faut fournir un objet – le builder – qui accepte des spécifications de votre nouvel objet étape par étape et qui gère tous les détails ennuyeux et complexes de l'instanciation. Puisque les objets builder gardent le contrôle de la configuration, ils peuvent vous aider à construire des objets valides. Un builder se trouve en position parfaite pour vérifier les actions du client et dire : "Non, je trouve qu'une cinquième roue sur une voiture ne serait pas très commode..."

Avec un peu d'ingéniosité, on peut créer des méthodes magiques pour faciliter le processus de construction. Pour y parvenir, on intercepte des appels de méthodes non définies avec `method_missing`, on analyse les noms des méthodes, et on construit le bon objet en fonction de ce nom. Les méthodes magiques représentent un pas en avant pour la construction rapide des objets, car le client est autorisé à spécifier des options de configuration multiples avec un appel de méthode unique.

Lorsque vous créez un builder et surtout lorsque vous l'utilisez, il faut se rendre compte du problème de sa réutilisation. Est-ce que vous pouvez utiliser une instance unique d'un builder pour créer des produits multiples ? Il est plus simple de créer des builders non réutilisables ou à réinitialiser lors d'une utilisation répétée plutôt que des builders complètement réutilisables. Il faut que vous vous posiez la question de savoir de quel type de Builder vous avez besoin.

Le pattern Builder est le dernier dans la famille des patterns de création d'objets[1] que nous examinerons dans cet ouvrage. Le chapitre suivant est consacré à la création de nouveaux objets, et nous allons retourner vers un autre sujet intéressant : la création d'un interpréteur.

1. Ou ceux qui empêchent la création, dans le cas du pattern Singleton.

15

Assembler votre système avec Interpreter

À la fin des années 1980, la version précédente de l'ingénieur logiciel Russ Olsen – probablement une version bêta – travaillait sur un système d'information géographique (GIS). L'un des objectifs clés de ce système GIS était sa capacité d'adaptation. Les cartes des utilisateurs étaient toutes différentes et chaque client avait sa propre vision de la présentation idéale d'une carte. Chaque client souhaitait utiliser ses cartes à sa guise et nous voulions naturellement que notre système puisse s'adapter à tous leurs caprices.

Malheureusement, le système était écrit en langage C et, malgré les nombreux points forts de ce langage, la capacité d'adaptation n'en fait pas partie. Écrire en C est difficile : il faut faire très attention à l'arithmétique des pointeurs sous peine de voir votre programme exploser en vol. Pire encore : le langage C se trouvait à un niveau conceptuel inadapté pour notre système. Lorsque vous écrivez un programme en C, vous gérez des `int`, des `float` et des pointeurs vers des `struct`, alors que nous voulions réfléchir en termes d'objets qui constituent des cartes tels que des vallées, des fleuves et des frontières politiques.

Les architectes de ce système GIS (un groupe d'élite dont je ne faisais pas partie) avaient alors résolu le problème de flexibilité par une décision radicale : le système ne devait pas être écrit en C. Environ 80 % de l'application était codée à l'aide d'un langage spécialisé qui savait gérer des notions géographiques telles que latitudes et longitudes. Ce langage proposait un langage de requêtes sophistiqué permettant d'effectuer simplement des opérations spécifiques, par exemple déplacer tous les arbres de taille moyenne 500 mètres plus au nord.

Ce langage de cartes ne permettait pas d'exprimer tout ce qu'un cartographe peut être amené à dire, mais il était néanmoins beaucoup plus adapté à cette tâche que n'importe quel programme en C.

80 % du système utilisait donc ce langage spécifique orienté carte. Et le reste ? Le reste était écrit en C. Cette partie écrite en C avait un objectif unique : fournir un interpréteur pour les 80 % du code écrit en langage spécialisé. Bref, ce vieux système GIS représentait une grande implémentation du pattern Interpreter.

Dans ce chapitre, nous allons nous familiariser avec le pattern Interpreter, qui nous enseigne que parfois le meilleur moyen de résoudre un problème est d'inventer un langage spécifique pour cette tâche. Nous allons explorer les différentes façons de développer un interpréteur classique et verrons quelques approches de la tâche fastidieuse d'analyse syntaxique. Nous apprendrons que les interpréteurs ne font pas partie des techniques de programmation les plus performantes, mais malgré le coût en terme de performances ils offrent beaucoup de flexibilité et de capacité d'extension.

Langage adapté à la tâche

L'idée fondamentale du pattern Interpreter est très simple : on peut résoudre certains problèmes de programmation en créant un langage spécifique capable d'exprimer la solution à nos problèmes. Quels problèmes sont de bons candidats pour le pattern Interpreter ? En règle générale, ces problèmes sont indépendants du reste de l'application et leur périmètre peut être clairement délimité. Ainsi, vous pourriez créer un langage de requêtes permettant de chercher des objets répondant à certaines spécifications[1]. Inversement, si votre problème nécessite la création de configurations complexes, considérez l'utilisation d'un langage de configuration.

Si vous écrivez des fragments de code assez simples, mais que vous soyez obligé de les arranger en combinaisons interminables, c'est un indicateur que le pattern Interpreter pourrait bien être la solution. Tout le travail de combinaison des modules pourrait être effectué par un interpréteur simple.

Développer un interpréteur

Le fonctionnement des interpréteurs se divise en deux phases. Premièrement, l'analyseur syntaxique parcourt le texte du programme pour produire un arbre de syntaxe abstrait (AST). L'AST représente la même information que le programme initial en

1. Cet exploit a déjà été réalisé pour des bases de données par les auteurs de SQL.

forme d'un arbre d'objets. Cela permet une exécution relativement efficace contrairement au programme initial en format textuel.

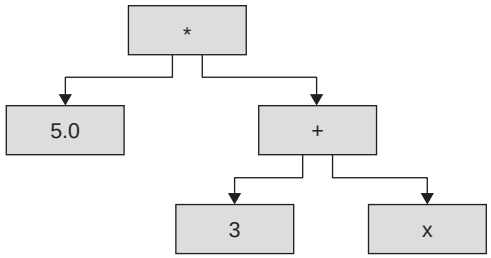


Figure 15.1
L'arbre syntaxique abstrait d'une simple expression arithmétique

Deuxièmement, l'AST s'évalue selon un ensemble de conditions externes, appelé contexte, pour produire le calcul requis.

On pourrait par exemple développer un interpréteur pour évaluer une expression arithmétique simple, telle que :

```
5.0*(3+x)
```

Tout d'abord, il faut analyser l'expression. L'analyseur syntaxique commence par le premier caractère de l'expression : le chiffre 5. Il poursuit vers la virgule et le zéro décimal, qui indiquent que c'est un nombre à virgule flottante. Après avoir traité 5.0, l'analyseur continue le processus jusqu'à la fin de l'expression pour obtenir une structure de données (voir Figure 15.1).

La structure de données présentée à la Figure 15.1 représente notre AST. Les feuilles de l'AST – c'est-à-dire `5.0`, `3` et `x` – sont appelées des *nœuds terminaux*. Ils représentent les briques les plus élémentaires du langage. Les nœuds qui ne sont pas des feuilles – dans cet exemple `+` et `*` – sont (assez logiquement) nommés des *nœuds non terminaux*. Ils représentent des notions de niveau supérieur dans le langage.

Comme vous pouvez le voir sur le diagramme UML (voir Figure 15.2), les nœuds non terminaux possèdent une référence vers une ou plusieurs sous-expressions, ce qui nous permet de construire des arbres avec un nombre illimité de niveaux[1].

1. Oui, nous avons déjà vu ce diagramme. Un AST est effectivement un exemple spécialisé du pattern Composite dans lequel les nœuds non terminaux ont le rôle de composites.

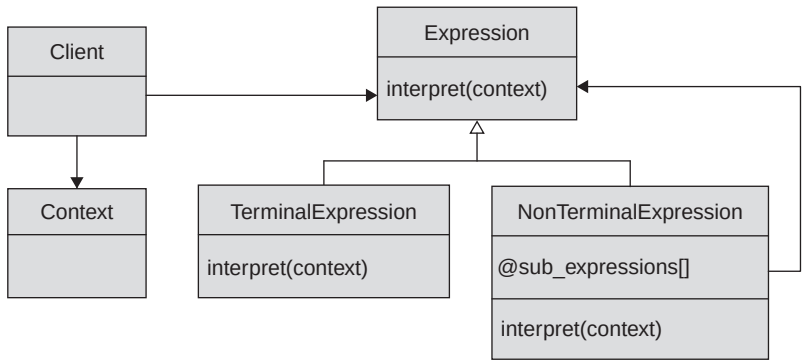


Figure 15.2
Diagramme de classe du pattern Interpreter

Bien que les membres du GoF aient nommé la méthode principale du pattern Interpreter `interpret`, des noms comme `evaluate` ou `execute` sont aussi appropriés et peuvent fréquemment être trouvés dans le code.

Une fois l'AST disponible, on pourrait évaluer notre expression, si ce n'était pour un détail mineur : quelle est la valeur de `x` ? Pour pouvoir évaluer l'expression il faut affecter une valeur à `x`. Est-ce 1, 167 ou –279 ? Les membres du GoF appellent *contexte* les valeurs ou les conditions fournies au moment de l'interprétation de l'AST. Retournons à notre exemple. Si l'on évalue notre AST avec `x` égal à `1`, le résultat obtenu est `20.0` ; si l'on évalue l'expression encore une fois avec la valeur de `x` à `4`, le résultat serait égal à `35.0`.

Quelle que soit la valeur du contexte, l'AST exécute l'évaluation en parcourant l'arbre de manière récursive. On demande au nœud à la racine de l'arbre de s'auto-évaluer (dans notre cas, c'est le nœud qui représente la multiplication). Ce nœud tente d'évaluer ses deux éléments récursivement. L'élément `5.0` ne présente pas de difficulté, mais le deuxième élément, l'opérateur d'addition, doit à son tour évaluer ses éléments (`3` et `x`). Enfin, nous arrivons à la fin de l'arbre et les résultats des évaluations remontent comme une bulle d'air.

Ce simple exemple arithmétique nous apprend deux choses. Premièrement, l'implémentation du pattern Interpreter est assez complexe. Souvenez-vous de toutes les classes qui constituent l'AST ainsi que l'analyseur syntaxique. L'envergure même de l'implémentation limite l'utilisation du pattern Interpreter à des langages relativement simples : on essaie de résoudre un problème réel et non pas de s'aventurer dans la recherche informatique. La deuxième conclusion que l'on puisse tirer est la performance limitée de ce pattern. Le besoin de parcourir l'AST, sans mentionner l'analyse syntaxique, pénalise la vitesse d'exécution.

En échange de cette complexité et de cette moindre performance, un interpréteur offre quelques avantages dont le principal est la flexibilité. Dès que l'interpréteur est disponible, il est très simple d'y ajouter des opérations. Il est assez facile d'imaginer qu'une fois notre petit interpréteur d'expressions arithmétiques prêt, il sera facile de rajouter des nœuds de soustraction et de multiplication dans l'AST. Un AST est une structure de données représentant un fragment spécifique de logique de programmation. Notre AST était écrit initialement pour évaluer cette logique et on peut le modifier pour prendre en charge d'autres tâches. On pourrait par exemple demander à l'AST d'afficher sa description :

```
Multiply 5.0 by the sum of 3 and x, where x is 1.
```

Un interpréteur pour trouver des fichiers

Assez de théorie, créons un interpréteur Ruby. Redévelopper un nouvel interpréteur de calcul arithmétique est probablement la dernière chose dont le monde a besoin et nous allons donc tenter quelque chose de différent. Écrivons un outil qui nous permet de gérer de grandes quantités de fichiers de formats et de tailles variables. Pour ce faire, nous serons fréquemment amenés à faire des recherches sur l'ensemble des fichiers selon des critères bien précis : par exemple rechercher tous les fichiers MP3 ou tous les fichiers disponibles en écriture. Qui plus est, nous souhaitons aussi retrouver les fichiers qui répondent à une combinaison de critères, par exemple tous les fichiers MP3 de grande taille ou tous les fichiers JPEG protégés en écriture.

Retrouver tous les fichiers

Cela ressemble fort à un problème qu'on pourrait résoudre à l'aide d'un langage de requêtes simple. Supposons que chaque expression de notre langage spécifie le type de fichier recherché. Commençons par les éléments de l'AST, l'analyseur syntaxique viendra plus tard.

La recherche la plus basique retourne tout simplement la totalité des fichiers. Définissons la classe pour accomplir cette tâche :

```
require 'find'
class Expression
  # Le code des expressions fréquentes sera bientôt inclus ici...
end
class All < Expression
  def evaluate(dir)
    results= []
```

```
      Find.find(dir) do |p|
        next unless File.file?(p)
        results << p
      end
      results
    end
  end
```

Fonctionnellement parlant, ce code n'est pas très riche. La méthode clé de notre interpréteur est nommée `evaluate`. La méthode `evaluate` de la classe `All` s'appuie tout simplement sur la classe `Find` de la bibliothèque standard de Ruby, qui permet de récupérer tous les fichiers résidant dans un dossier donné. Si l'on passe à la méthode `Find.find` un nom du dossier et un bloc, le bloc en question est exécuté pour chaque élément du dossier. J'insiste sur le mot "chaque". Vu que la méthode `Find` agit de manière récursive, le bloc serait appelé non seulement sur chacun des fichiers du dossier mais aussi sur tous les sous-dossiers, tous les fichiers des sous-dossiers et ainsi de suite. Dans notre cas, seuls les fichiers nous intéressent. Il faut donc mettre en place un mécanisme de filtrage. La ligne

```
next unless File.file?(p)
```

ignore tout élément qui n'est pas un fichier.

Ne vous inquiétez pas au sujet de la classe parent `Expression` vide. Nous allons y ajouter du code fort utile très prochainement.

Rechercher des fichiers par nom

L'étape suivante tombe sous le sens : nous allons créer une classe chargée de retourner tout fichier dont le nom correspond à un patron donné :

```
class FileName < Expression
  def initialize(pattern)
    @pattern = pattern
  end

  def evaluate(dir)
    results= []
    Find.find(dir) do |p|
      next unless File.file?(p)
      name = File.basename(p)
      results << p if File.fnmatch(@pattern, name)
    end
    results
  end
end
```

La classe `FileName` est légèrement plus compliquée que la classe `All`. Elle fait appel à quelques méthodes très utiles de la classe `File`. La méthode `File.basename` retourne la partie du chemin qui correspond au nom d'un fichier : passez `"/home/russ/chapter1.doc"` à cette méthode et vous obtiendrez `"chapter1.doc"`. La méthode `File.fnmatch` renvoie `true` uniquement si le patron du nom des fichiers spécifié dans le premier paramètre (par exemple `"*.doc"`) correspond au nom du fichier passé dans le deuxième paramètre (par exemple `"chapter1.doc"`).

Nos classes de recherche sont très simples d'utilisation. Si le dossier `test_dir` contient deux fichiers MP3 et une image, nous pouvons récupérer les trois fichiers à l'aide du code suivant :

```
expr_all = All.new
files = expr_all.evaluate('test_dir')
```

Mais, si l'on s'intéresse uniquement aux MP3, on peut faire ceci :

```
expr_mp3 = FileName.new('*.mp3')
mp3s = expr_mp3.evaluate('test_dir')
```

Dans l'exemple précédent, le nom du dossier passé à la méthode `evaluate` joue le rôle du contexte, c'est-à-dire les paramètres extérieurs qui servent à l'évaluation de l'expression. La même expression peut être évaluée avec des contextes différents, ce qui engendre des résultats différents. On pourrait, par exemple, rechercher tous les fichiers MP3 dans le dossier `music_dir` :

```
other_mp3s = expr_mp3.evaluate('music_dir')
```

Des grands fichiers et des fichiers ouverts en écriture

Il est clair que la recherche de fichiers correspondant à un certain nom est loin d'être la seule option de recherche possible. On pourrait par exemple avoir besoin de retrouver tous les fichiers dont la taille est supérieure à une valeur donnée :

```
class Bigger < Expression
  def initialize(size)
    @size = size
  end

  def evaluate(dir)
    results = []
    Find.find(dir) do |p|
      next unless File.file?(p)
      results << p if( File.size(p) > @size)
    end
    results
  end
end
```

Ou on pourrait rechercher les fichiers disponibles en écriture :

```
class Writable < Expression
  def evaluate(dir)
    results = []
    Find.find(dir) do |p|
      next unless File.file?(p)
      results << p if( File.writable?(p) )
    end
    results
  end
end
```

Des recherches plus complexes à l'aide des instructions Not, And et Or

Désormais, nous avons quelques classes de base pour rechercher des fichiers et ces classes constituent les nœuds terminaux de notre AST. Passons aux choses plus intéressantes. Que faire si l'on souhaite retrouver tous les fichiers protégés en écriture ? On pourrait évidemment définir encore une classe qui ressemble à celles qui existent déjà. Mais essayons une autre option : définissons notre premier nœud non terminal, en l'occurrence le nœud `Not` :

```
class Not < Expression
  def initialize(expression)
    @expression = expression
  end

  def evaluate(dir)
    All.new.evaluate(dir) - @expression.evaluate(dir)
  end
end
```

Le constructeur de la classe `Not` accepte un argument qui représente l'expression que nous souhaitons évaluer en négatif. Lorsque la méthode `evaluate` est appelée, elle commence par retrouver la totalité des chemins à l'aide de la classe `All` et exécute ensuite la méthode de soustraction de la classe `Array` pour supprimer les chemins des fichiers retournés par l'expression. Nous nous retrouvons à la fin avec l'ensemble des chemins qui ne répondent pas à l'expression. Par conséquent, pour retrouver tous les fichiers non disponibles en écriture on écrirait :

```
expr_not_writable = Not.new( Writable.new )
readonly_files = expr_not_writable.evaluate('test_dir')
```

La classe `Not` est très élégante car elle ne s'applique pas seulement à `Writable`. On pourrait choisir d'employer `Not` pour trouver tous les fichiers dont la taille est inférieure à 1 Ko :

```
small_expr = Not.new( Bigger.new(1024) )
small_files = small_expr.evaluate('test_dir')
```

ou rechercher tous les fichiers dont le format n'est pas MP3 :

```
not_mp3_expr = Not.new( FileName.new('*.mp3') )
not_mp3s = not_mp3_expr.evaluate('test_dir')
```

Il est aussi possible de définir un nœud non terminal pour combiner deux expressions de recherche :

```
class Or < Expression
  def initialize(expression1, expression2)
    @expression1 = expression1
    @expression2 = expression2
  end

  def evaluate(dir)
    result1 = @expression1.evaluate(dir)
    result2 = @expression2.evaluate(dir)
    (result1 + result2).sort.uniq
  end
end
```

La classe `Or` nous permet de récupérer en une seule requête tous les fichiers au format MP3 *ou* ceux dont la taille est supérieure à 1 Ko :

```
big_or_mp3_expr = Or.new( Bigger.new(1024), FileName.new('*.mp3') )
big_or_mp3s = big_or_mp3_expr.evaluate('test_dir')
```

La classe `Or` indique que `And` ne doit pas être bien loin :

```
class And < Expression
  def initialize(expression1, expression2)
    @expression1 = expression1
    @expression2 = expression2
  end

def evaluate(dir)
    result1 = @expression1.evaluate(dir)
    result2 = @expression2.evaluate(dir)
    (resultl & result2)
  end
end
```

Désormais, nous avons sous la main tous les outils pour spécifier des recherches de fichiers complexes. Tentons de récupérer tous les MP3 de grande taille protégés en écriture :

```
complex_expression = And.new(
  And.new(Bigger.new(1024),
    FileName.new('*.mp3')),
  Not.new(Writable.new))
```

Cette expression complexe nous révèle une autre propriété intéressante du pattern Interpreter. Une fois que nous avons défini un AST complexe comme le précédent, il peut être réutilisé dans des contextes différents :

```
complex_expression.evaluate('test_dir')
complex_expression.evaluate('/tmp')
```

Conçu correctement, un pattern Interpreter vous récompensera généreusement pour vos efforts. Dans notre exemple, seules sept classes auront été nécessaires pour obtenir un AST de recherche de fichiers relativement flexible.

Créer un AST

De façon assez surprenante, le pattern Interpreter défini par le GoF reste muet sur la création de l'AST lui-même. Le pattern suppose que l'AST est déjà disponible et omet l'étape de son développement. Il est pourtant assez évident qu'un AST doit être créé et, pour cela, nous disposons d'une large palette de possibilités.

Un analyseur syntaxique simple

La manière probablement la plus évidente d'obtenir un AST consiste à développer un analyseur syntaxique. Créer un analyseur syntaxique pour notre langage de recherche de fichiers ne présente pas de difficultés particulières. Supposons que la syntaxe ressemble à ceci :

```
and (and(bigger 1024)(filename *.mp3)) writable
```

Le code suivant (de 50 lignes environ) effectue correctement notre analyse syntaxique :

```
class Parser
  def initialize(text)
    @tokens = text.scan(/\(|\)|[\w\.\*]+/)
  end

  def next_token
    @tokens.shift
  end

  def expression
    token = next_token
    if token == nil
      return nil
    elsif token == '('
      result = expression
      raise 'Expected )' unless next_token == ')
      result
    elsif token == 'all'
      return All.new
```

```
      elsif token == 'writable'
        return Writable.new
      elsif token == 'bigger'
        return Bigger.new(next_token.to_i)
      elsif token == 'filename'
        return FileName.new(next_token)
      elsif token == 'not'
        return Not.new(expression)
      elsif token == 'and'
        return And.new(expression, expression)
      elsif token == 'or'
        return Or.new(expression, expression)
      else
        raise "Unexpected token: #{token}"
      end
    end
  end
```

Pour utiliser cet analyseur syntaxique il suffit de passer les expressions de recherche de fichiers au constructeur et d'appeler la méthode `parse`. Cette méthode retourne l'AST correspondant, prêt à être utilisé :

```
parser = Parser.new "and (and(bigger 1024)(filename *.mp3)) writable"
ast = parser.expression
```

La classe `Parser` applique la méthode `scan` de la classe `String` pour diviser l'expression en fragments convenables :

```
@tokens = text.scan(/\(|\)|[\w\.\*]+/)
```

La chaîne est divisée en un tableau de sous-chaînes appelées tokens par la magie des expressions régulières[1]. Chacun des tokens est soit une parenthèse soit un fragment de texte contigu comme "filename" ou "*.mp3"[2].

La plus grande partie de l'analyseur syntaxique est occupée par la méthode `expression`. Elle parcourt les tokens un par un afin de construire l'AST.

---

1. Si vous n'êtes pas familier avec les expressions régulières, jetez un œil sur l'Annexe B, Aller plus loin. Les expressions régulières valent bien la peine d'être étudiées.
2. Si vous connaissez les expressions régulières, vous avez sans doute remarqué que mon analyseur syntaxique ne gère pas les noms de fichiers qui contiennent des espaces. Je me permets de répéter que les exemples doivent rester simples, sans parler du fait que la pratique d'incorporer des espaces dans les noms des fichiers est à mon avis contraire à l'éthique.

Et un interpréteur sans analyseur ?

Même si le développement de ce premier analyseur syntaxique n'a pas présenté de difficultés particulières, un certain effort a cependant été nécessaire et on peut donc se poser la question de la nécessité d'un tel analyseur. Les classes de recherche de fichiers que nous avons réalisées peuvent elles-mêmes constituer une bonne API interne orientée programmeur. Si le but est de pouvoir spécifier un bon moyen de déclencher des recherches de fichiers à partir de notre code, on pourrait probablement créer dans le code un AST de recherche de fichiers de la même façon que dans les exemples de la section précédente. Cela nous permettrait de profiter de toute la flexibilité et de l'extensibilité du pattern Interpreter sans devoir gérer l'analyse syntaxique.

Si vous décidez d'opter pour un interpréteur sans analyseur, pensez à rajouter quelques noms de méthodes raccourcis pour simplifier la vie de vos utilisateurs. Nous pouvons par exemple étendre notre interpréteur de recherche de fichiers en définissant dans la classe `Expression` quelques opérateurs pour créer des expressions `And` et `Or` avec une syntaxe plus compacte[1] :

```
class Expression
  def |(other)
    Or.new(self, other)
  end

  def &(other)
    And.new(self, other)
  end
end
```

On se doutait un peu que la classe `Expression` finirait bien par être utilisée ! Grâce aux opérateurs nous pouvons lancer des recherches de fichiers complexes sans taper des longues expressions au clavier. Au lieu de

```
Or.new(
  And.new(Bigger.new(2000), Not.new(Writable.new)),
  FileName.new('*.mp3'))
```

nous pouvons écrire

```
(Bigger.new(2000) & Not.new(Writable.new)) | FileName.new("*.mp3")
```

1. Malgré le fait qu'il n'y a pas de contre-indications des opérateurs, cette pratique doit être utilisée avec modération, car un nombre d'opérateurs excessif a tendance à rendre le code difficilement lisible.

Nous pouvons aller plus loin encore dans la simplification de la syntaxe. Il suffit de définir des méthodes raccourcies pour créer des nœuds terminaux :

```
def all
  All.new
end

def bigger(size)
  Bigger.new(size)
end

def name(pattern)
  FileName.new(pattern)
end

def except(expression)
    Not.new(expression)
end

def writable
  Writable.new
end
```

Ces nouvelles méthodes réduisent encore davantage l'expression de recherche précédente :

```
(bigger(2000) & except(writable) ) | file_name('*.mp3')
```

Il ne faut pas oublier que nous ne pouvons pas utiliser le nom `not` pour raccourcir l'appel `Not.new` à cause du conflit de nom avec l'opérateur `not` de Ruby.

Déléguer l'analyse à XML ou à YAML ?

Si toutefois vous jugez qu'un analyseur syntaxique est nécessaire, il existe une alternative assez intéressante au développement de votre propre analyseur qui consiste à s'appuyer sur XML ou sur YAML[1]. Cette solution vous permet de vous fier aux bibliothèques d'analyse syntaxique de XML ou de YAML qui sont livrées avec votre distribution de Ruby. Au premier abord, l'idée paraît formidable : vous profitez ainsi de toute la flexibilité et des possibilités d'extension d'un interpréteur complet sans vous soucier des détails d'analyse syntaxique. Rien à dire, n'est-ce pas ?

Malheureusement, cette idée pourrait bien provoquer quelques plaintes de la part de vos utilisateurs. Alors que XML et YAML sont très bien adaptés pour représenter des données, ils ne sont pas idéals pour exprimer des programmes. Gardez à l'esprit que le

1. YAML ou "YAML Ain't Markup Language" est un format en texte simple. Tout comme XML il est utilisé pour stocker des données hiérarchiques. Contrairement à XML, YAML est très convivial et il est très populaire dans la communauté Ruby.

développement d'un interpréteur est motivé principalement par la volonté de proposer à vos utilisateurs un moyen naturel pour exprimer leurs besoins de traitement. Si les idées encapsulées dans votre interpréteur peuvent être exprimées naturellement en XML ou YAML, n'hésitez pas à opter pour ce genre de formats afin de profiter pleinement des analyseurs syntaxiques fournis. Mais, si votre langage ne correspond pas vraiment à ces formats – et je me permets d'affirmer que c'est le cas de la majorité des langages du pattern Interpreter –, n'essayez pas de forcer votre langage à entrer dans un format inadapté par pure paresse.

Racc pour des analyseurs plus complexes

Si votre langage est relativement complexe et que ni XML ni YAML ne semblent appropriés, vous pouvez opter pour l'utilisation d'un générateur d'analyseurs syntaxiques tel que Racc. Racc hérite son modèle (et son nom) du vénérable utilitaire UNIX : YACC. Racc prend en entrée la description de la grammaire propre à votre langage et génère un analyseur syntaxique Ruby pour ce langage. Racc est un outil formidable mais âmes sensibles s'abstenir : apprendre à se servir d'un générateur d'analyseurs syntaxiques est long et la courbe d'apprentissage est raide.

Déléguer l'analyse à Ruby ?

Il existe une autre réponse au dilemme de l'analyseur syntaxique. Vous pourriez décider d'implémenter votre pattern Interpreter de façon à permettre aux utilisateurs d'écrire du code Ruby classique. Il est peut-être possible de concevoir l'API de votre AST de façon que le code s'insère naturellement dans le reste du code Ruby. Ainsi, vos utilisateurs ne sauraient même pas qu'ils écrivent du code Ruby. Cette idée est tellement curieuse que le chapitre suivant lui est entièrement consacré.

User et abuser du pattern Interpreter

Le pattern Interpreter a tendance à être sous-utilisé : à mon avis, cette caractéristique le distingue des autres patterns du GoF exposés dans ce livre. Pendant ma carrière j'ai rencontré un certain nombre de systèmes qui auraient pu profiter du pattern Interpreter. Ces systèmes investissaient beaucoup d'effort dans des solutions fondées sur des conceptions pas complètement adéquates. Par exemple, à l'âge de pierre des bases de données une requête se présentait comme un programme codé laborieusement par un expert des bases de données. Cette approche a perduré pendant une longue période jusqu'à l'apparition des langages (en majorité interprétés) tels que SQL. De la même manière, pendant des années la construction d'une interface graphique même la plus

simple nécessitait l'intervention d'un ingénieur logiciel. Il devait passer des jours et des semaines à écrire du code page après page. Aujourd'hui, tout collégien qui a accès à un clavier peut développer des interfaces graphiques relativement complexes à l'aide d'un langage interprété que nous appelons HTML.

Pourquoi le pattern Interpreter est-il négligé ? De nombreux ingénieurs logiciels qui passent leurs journées à développer des solutions métier sont souvent des experts en conception de bases de données ou en développement d'applications Web, mais la dernière fois où ils ont vu des AST ou des analyseurs syntaxiques remonte probablement à leur deuxième année d'école d'ingénieurs. C'est regrettable. Comme nous l'avons vu, une application correcte du pattern Interpreter donne à votre système une flexibilité redoutable.

Nous avons déjà noté quelques inconvénients majeurs des interpréteurs. Tout d'abord, ils sont complexes. Lorsque vous considérez l'usage du pattern Interpreter, surtout si vous planifiez de construire un analyseur syntaxique, essayez d'estimer la complexité de votre langage. Cherchez à le rendre le plus simple possible. Réfléchissez au profil des futurs utilisateurs de votre langage. Seront-ils des ingénieurs logiciel expérimentés capables d'avancer avec le minimum de messages d'erreurs ? Ou s'agira-t-il d'utilisateurs moins techniques qui auront besoin d'un diagnostic détaillé en cas de problème ?

Par ailleurs, le problème de l'efficacité du programme se pose aussi. N'oubliez pas que la vitesse d'exécution de l'expression

```
Add.new(Constant.new(2), Constant.new(2)).interpret
```

ne sera jamais comparable à celle de

```
2 + 2
```

Même avec toute sa flexibilité et sa puissance, le pattern Interpreter ne sera jamais un bon choix pour les 2 % de votre code dont le temps d'exécution est critique. Mais pourquoi ne pas l'appliquer aux 98 % du code restant ?

Des interpréteurs dans le monde réel

Il est facile de trouver des exemples d'interpréteurs dans le monde Ruby. Le langage Ruby lui-même est évidemment un langage interprété, quoique légèrement plus complexe que celui prévu par le pattern Interpreter.

De la même manière, les expressions régulières – des outils formidables pour comparer du texte à des patrons, qui ont été si utiles pour notre implémentation d'un analyseur syntaxique – sont elles-mêmes implémentées sous forme d'un langage interprété.

Lorsque vous écrivez l'expression régulière `/[rR]uss/`, elle est traduite en un AST de façon transparente pour rechercher des occurrences des deux variantes de mon prénom.

Il existe également Runt[1] : une bibliothèque qui fournit un langage simple pour exprimer des intervalles de temps et de dates ainsi que des calendriers. Avec Runt on peut écrire des expressions temporelles qui ne correspondront qu'à certains jours de la semaine :

```
require 'rubygems'
require 'runt'
mondays = Runt::DIWeek.new(Runt::Monday)
wednesdays = Runt::DIWeek.new(Runt::Wednesday)
fridays = Runt::DIWeek.new(Runt::Friday)
```

Les trois objets ci-dessus nous permettent de découvrir que Noël en 2015 tombera un vendredi, car le code

```
fridays.include?(Date.new(2015,12,25))
```

retourne `true`, alors que

```
mondays.include?(Date.new(2015,12, 25))
wednesdays.include?(Date.new(2015,12,25))
```

retournent `false`.

Comme d'habitude avec le pattern Interpreter, vous utilisez pleinement la puissance de Runt lorsque vous commencez à combiner des expressions. Voici une expression Runt qui exprime l'emploi du temps horrible que j'ai subi à l'université :

```
nine_to_twelve = Runt::REDay.new(9,0,12,0)
class_times = (mondays | wednesdays | fridays) & nine_to_twelve
```

Runt est un bon exemple d'interpréteur sans analyseur syntaxique : il est conçu comme une simple bibliothèque de classes pour des programmeurs Ruby.

En conclusion

Dans ce chapitre, nous avons découvert le pattern Interpreter. Ce pattern suggère que parfois un interpréteur simple est le meilleur moyen de résoudre un problème. Le pattern Interpreter est bien adapté aux problèmes clairement délimités tels que des langages de requête ou de configuration. C'est également une bonne approche pour combiner des blocs de fonctionnalités.

---

1. La bibliothèque Runt a été écrite par Matthew Lipper. Elle repose sur la notion d'expressions temporelles de Martin Fowler.

L'arbre de syntaxe abstraite est le cœur du pattern Interpreter. Vous traitez votre langage spécialisé comme une série d'expressions que vous décomposez en une structure arborescente. À vous de choisir comment effectuer cette décomposition : on peut fournir aux clients une API pour construire l'arbre dans le code ou on peut implémenter un analyseur syntaxique qui accepte des chaînes de caractères pour les convertir en un AST. Dans les deux cas, une fois l'AST disponible, il peut s'autoévaluer pour retourner un résultat.

La flexibilité et l'extensibilité sont les principaux atouts du pattern Interpreter. Les mêmes classes de l'interpréteur peuvent accomplir des tâches différentes en fonction des AST. Étendre votre langage pour ajouter de nouveaux nœuds dans l'AST est généralement assez simple. Toutefois, ces avantages entraînent un coût en termes de performance et de complexité. Les interpréteurs ont tendance à être lents et il est difficile d'en accélérer l'exécution. Il est donc recommandé de limiter leur usage aux modules qui ne nécessitent pas une grande rapidité d'exécution. Cette complexité est la conséquence de l'infrastructure nécessaire à la mise en œuvre du pattern Interpreter : il faut implémenter toutes les classes pour constituer un AST et, possiblement, l'analyseur.

Dans le chapitre suivant, nous allons étudier les langages spécifiques d'un domaine (DSL). C'est un pattern qui est étroitement lié au pattern Interpreter. Nous allons nous concentrer particulièrement sur des DSL internes : une alternative élégante au travail (parfois pénible) d'implémentation d' un analyseur syntaxique pour votre interpréteur.

Partie III

Les patterns Ruby

16

Ouvrir votre système avec des langages spécifiques d'un domaine

Au Chapitre 15, nous avons examiné comment résoudre certains types de problèmes à l'aide des interpréteurs. L'arbre de syntaxe abstrait (AST) qui est employé pour obtenir un résultat ou effectuer une action constitue l'élément clé du pattern Interpreter. Nous avons découvert au chapitre précédent que le pattern Interpreter n'est pas directement concerné par la création de l'AST. On suppose que l'arbre est disponible et on se concentre sur sa façon de fonctionner. Dans ce chapitre, nous allons explorer le pattern Domain-Specific Language (DSL), qui voit le monde par l'autre bout de la lorgnette. Le pattern DSL stipule qu'il faut diriger son attention vers le langage même et non pas vers l'interpréteur. Il est parfois possible de faciliter la résolution d'un problème en fournissant aux utilisateurs une syntaxe adaptée.

Vous ne trouverez pas le pattern DSL dans le *Design Patterns* du GoF. Néanmoins, comme vous le verrez dans ce chapitre, l'incroyable flexibilité de Ruby permet d'implémenter très simplement un style particulier de DSL.

Langages spécifiques d'un domaine

Le pattern DSL n'est pas différent de la majorité des autres patterns couverts dans ce livre : ici comme ailleurs, l'idée fondatrice n'est pas très compliquée. On comprend le principe des DSL lorsqu'on prend du recul et qu'on se demande ce qu'on cherche à atteindre en écrivant des programmes. La réponse est (je l'espère) de rendre nos utilisateurs heureux. Un utilisateur veut se servir d'un ordinateur pour accomplir une tâche :

gérer des comptes financiers ou diriger une sonde spatiale vers Mars. Bref, l'utilisateur souhaite que l'ordinateur satisfasse une demande. On pourrait alors se poser une question naïve : pourquoi l'utilisateur a-t-il besoin de nous ? Pourquoi ne pas lui passer l'interpréteur Ruby en lui souhaitant bonne chance ? C'est une idée ridicule car, en règle générale, les utilisateurs ne comprennent pas la programmation et les ordinateurs. Ils maîtrisent des bits et des octets aussi bien que nous maîtrisons la comptabilité et la mécanique céleste. L'utilisateur connaît sa matière, son domaine, mais pas le domaine de la programmation.

Et si l'on pouvait créer un langage de programmation qui permettrait à un utilisateur d'exprimer certaines des règles métier qui concernent son domaine spécifique au lieu de règles compliquées intrinsèquement liées aux ordinateurs ? Des comptables pourraient alors faire appel à des notions de comptabilité et des chercheurs en aérospatiale pourraient parler de sondes. Dans ce cas, l'idée de fournir un langage à l'utilisateur ne paraît plus si absurde.

Il est assurément possible de développer ce type de langages en appliquant des techniques apprises au Chapitre 15. On pourrait retrousser nos manches et concevoir un analyseur syntaxique pour un langage de comptabilité ou recourir à Racc pour créer un langage de navigation céleste. Martin Fowler nomme ces approches plus ou moins traditionnelles des DSL externes. Ces langages sont externes dans le sens où ils contiennent deux entités complètement distinctes. D'un côté, il y a l'analyseur syntaxique et l'interpréteur du langage, de l'autre côté, il y a des programmes écrits en ce langage. Si l'on créait un DSL spécialisé pour des comptables avec l'analyseur et l'interpréteur écrits en Ruby, on se retrouverait avec deux choses complètement séparées : le DSL de comptabilité d'une part et le programme pour l'interpréter d'autre part.

Étant donné l'existence des DSL externes, on pourrait se demander s'il existe des langages internes et quelles sont les différences entre les deux. Selon Martin Fowler, un DSL interne consiste à partir d'un langage d'implémentation connu – par exemple Ruby – pour le transformer en DSL. Si l'on utilise Ruby pour implémenter notre DSL (si vous avez vu le titre de ce livre, vous vous doutez que c'est le cas), tout utilisateur qui écrit un programme en utilisant notre petit langage produit en réalité et probablement sans le savoir un programme Ruby.

Un DSL pour des sauvegardes de fichiers

Il s'avère que construire un DSL interne à Ruby est assez simple. Imaginez que nous devions mettre en place un programme de sauvegarde : un système qui s'exécuterait à

intervalles réguliers pour copier nos fichiers importants dans un autre répertoire (vraisemblablement mieux protégé). On décide de réaliser cet exploit à l'aide d'un DSL, un langage nommé PackRat qui permettrait aux utilisateurs d'exprimer quels fichiers il faut sauvegarder et à quel moment. On pourrait avoir la syntaxe suivante :

```
backup '/home/russ/documents'
backup '/home/russ/music', file_name('*.mp3') & file_name('*.wav')
backup '/home/russ/images', except(file_name('*.tmp'))
to '/external_drive/backups'
interval 60
```

Ce petit programme écrit en PackRat déclare que nous avons trois dossiers pleins de données que nous voulons copier dans le répertoire `/external_drive/backups` une fois par heure (toutes les 60 minutes). Il faudrait sauvegarder la totalité du dossier `documents`, ainsi que la totalité du dossier `images` à l'exception des fichiers temporaires. En ce qui concerne le dossier `music`, nous voulons seulement copier des fichiers audio. Comme nous avons horreur de réinventer des choses qui existent déjà, PackRat fait appel à des expressions de recherche de fichiers que nous avons déjà développées au Chapitre 15.

C'est un fichier de données, non, c'est un programme !

Pour attaquer ce projet PackRat nous pourrions sortir nos expressions régulières favorites ou un générateur d'analyseurs syntaxiques pour produire un analyseur traditionnel : le premier mot lu doit être "backup", puis on recherche un guillemet, etc. Mais il doit y avoir une solution plus simple. En regardant bien on se rend compte que les instructions `backup` pourraient presque être des appels de méthodes Ruby. Stop ! Ils peuvent être des appels de méthodes Ruby. Si `backup`, `to` et `interval` étaient des noms de méthodes Ruby, ce code représenterait un programme Ruby parfaitement valide, c'est-à-dire une série d'appels de `backup`, `to` et `interval`, chacun avec un ou deux arguments. Les parenthèses autour des arguments sont omises, mais c'est évidemment parfaitement acceptable en Ruby.

Pour commencer, essayons d'écrire un petit programme Ruby qui ne fait rien d'autre que lire le fichier `backup.pr file`. Voici un petit programme qui s'appelle `packrat.rb`, c'est le début de notre interpréteur DSL :

```
require 'finder'
def backup(dir, find_expression=All.new)
  puts "Backup called, source dir=#{dir} find expr=#{find_expression}"
end
```

```
def to(backup_directory)
  puts "To called, backup dir=#{backup_directory}"
end

def interval(minutes)
  puts "Interval called, interval = #{minutes} minutes"
end
eval(File.read('backup.pr'))
```

Ce n'est pas un programme élaboré, mais ce code présente la plupart des idées nécessaires à la mise en œuvre d'un DSL interne en Ruby. Nous avons trois méthodes `backup`, `to` et `interval`. La partie clé du code de notre DSL est la dernière instruction :

```
eval(File.read('backup.pr'))
```

Cette expression commence la lecture du contenu de `backup.pr` et exécute ensuite ce contenu comme un programme Ruby[1]. Cela signifie qu'`interval`, `to` et toutes les expressions `backup` dans `backup.pr` – autrement dit tout ce qui ressemble à des méthodes Ruby – seront aspirés dans notre programme et interprétés comme des appels de méthodes Ruby. Lorsque l'on exécute `packrat.rb`, on obtient les messages de sortie de ces méthodes :

```
Backup called, source
dir=/home/russ/documents find expr=#<All:0xb7d84c14>
Backup called, source dir=/home/russ/music find expr=#<And:0xb7d84b74>
Backup called, source dir=/home/russ/images find expr=#<Not:0xb7d84afc>
To called, backup dir=/external_drive/backups
Interval called, interval = 60 minutes
```

Le qualificatif "interne" d'un DSL s'explique par cette technique d'aspiration et d'interprétation. L'expression `eval` fait fusionner l'interpréteur et le programme PackRat. C'est de la science-fiction sans peine.

Développer PackRat

Désormais, nos utilisateurs écrivent sans méfiance des appels de méthodes Ruby. Mais que devons-nous faire à l'intérieur de ces méthodes exactement ? Quel travail doivent effectuer `interval`, `to` et `backup` ? Ils doivent retenir le fait d'être appelés. Formulé autrement, ils doivent configurer certaines structures de données. Pour commencer, définissons la classe `Backup`, qui représente la totalité de la commande de sauvegarde :

1. Ruby fournit la méthode `load`, qui permet d'évaluer le contenu d'un fichier en tant que code Ruby en une étape, mais le traitement en deux étapes avec `read` et `eval` permet de mieux illustrer un DSL.

```
class Backup
  include Singleton
  attr_accessor :backup_directory, :interval
  attr_reader :data_sources
  def initialize
    @data_sources = []
    @backup_directory = '/backup'
    @interval = 60
  end

  def backup_files
    this_backup_dir = Time.new.ctime.tr(' :','_')
    this_backup_path = File.join(backup_directory, this_backup_dir)
    @data_sources.each { |source| source.backup(this backup path)}
  end

  def run
    while true
      backup_file
      sleep(@interval*60)
    end
  end
end
```

La classe `Backup` n'est qu'un conteneur pour l'information que contient le fichier backup.pr. Il déclare des attributs pour l'intervalle et le dossier cible de la sauvegarde ainsi qu'un tableau pour stocker les dossiers à sauvegarder. Le seul aspect légèrement plus complexe de la classe figure dans la méthode `run`. Cette méthode exécute des sauvegardes en copiant les données source dans le dossier de sauvegarde (qui est en réalité un sous-dossier estampillé du dossier de sauvegarde). Ensuite, elle se met en veille jusqu'à la sauvegarde suivante. La classe `Backup` est déclarée comme un singleton car notre utilitaire n'en aura jamais plusieurs.

Nous avons maintenant besoin d'une classe pour représenter les dossiers à sauvegarder :

```
class DataSource
  attr_reader :directory, :finder_expression
  def initialize(directory, finder_expression)
    @directory = directory
    @finder_expression = finder_expression
  end

  def backup(backup_directory)
    files=@finder_expression.evaluate(@directory)
    files.each do |file|
      backup_file( file, backup_directory)
    end
  end

  def backup_file(path, backup_directory)
    copy_path = File.join(backup_directory, path)
```

```
      FileUtils.mkdir_p(File.dirname(copy_path))
      FileUtils.cp(path, copy_path)
    end
  end
```

La classe `DataSource` est le conteneur d'un chemin vers un dossier et de l'AST des expressions de recherche de fichiers. Elle contient également la plupart de la logique nécessaire à la copie des fichiers.

Assembler notre DSL

Puisque la totalité du code utilitaire est disponible, faire fonctionner le DSL PackRat devient un jeu d'enfant. Nous allons réécrire les méthodes initiales `backup`, `to` et `interval` afin qu'elles utilisent les classes que nous venons de créer :

```
def backup(dir, find_expression=All.new)
  Backup.instance.data_sources << DataSource.new(dir, find_expression)
end

def to(backup_directory)
  Backup.instance.backup_directory = backup_directory
end

def interval(minutes)
  Backup.instance.interval = minutes
end
eval(File.read('backup.pr'))
Backup.instance.run
```

Nous allons examiner ce code méthode par méthode. La méthode `backup` ne fait que récupérer l'instance du singleton `Backup` et lui ajouter une source de données. De la même manière, la méthode `interval` récupère la valeur de l'intervalle de sauvegarde et l'affecte au champ correspondant du singleton `Backup`. La méthode `to` fait la même action pour le chemin du dossier de sauvegarde.

Enfin, nous avons deux dernières lignes qui terminent notre interpréteur `PackRat` :

```
eval(File.read('backup.pr'))
Backup.instance.run
```

L'expression `eval` nous est déjà familière : elle déclenche la lecture du fichier PackRat ainsi que son évaluation en tant que code Ruby. La toute dernière ligne du programme démarre le processus de sauvegarde.

La structure de l'interpréteur PackRat est assez typique d'un DSL interne. On commence par définir les structures de données : dans notre cas, c'est la classe `Backup` et compagnie. Ensuite, on définit quelques méthodes de niveau supérieur pour supporter le

langage DSL même : dans le cas de PackRat, ce sont les méthodes `interval`, `to` et `backup`. Puis on aspire le texte du DSL à l'aide de l'instruction `eval(File.read(...))`. L'importation du texte DSL initialise les structures de données. Dans notre cas, nous nous retrouvons avec une instance de `Backup` complètement configurée. Enfin, on fait ce que l'utilisateur nous demande de faire : toutes les instructions sont disponibles dans nos structures de données initialisées.

Récolter les bénéfices de PackRat

L'approche d'un DSL interne offre quelques avantages : nous avons réussi à créer un vrai DSL de sauvegarde en moins de 70 lignes de code. Une grande partie de ce code est dédiée à l'infrastructure `Backup`/`Source`, qui serait probablement nécessaire quel que soit le choix de l'implémentation. L'atout supplémentaire d'un DSL interne fondé sur Ruby réside dans l'accès gratuit à la totalité de l'infrastructure du langage. Si vous aviez un nom de dossier comprenant un guillemet simple[1], vous pourriez échapper ce caractère comme vous le faites habituellement dans Ruby :

```
backup '/home/russ/bob\'s_documents'
```

Puisque c'est Ruby, vous pourriez également faire ceci :

```
backup "/home/russ/bob's_documents"
```

Si l'on écrivait une implémentation classique d'un analyseur syntaxique, on serait obligé de gérer ce guillemet imbriqué. Ce n'est pas le cas ici, car on hérite cette fonctionnalité de Ruby. De la même manière, nous obtenons gratuitement des commentaires :

```
#
# Sauvegarder le dossier de Bob
#
backup "/home/russ/bob's_documents"
```

En cas de besoin, nos utilisateurs peuvent profiter de toutes les possibilités de programmation proposées par Ruby :

```
#
# Une expression de recherche de fichiers audio
#
music_files = file_name('*.mp3') | file_name('*.wav')
```

1. Pour moi, un tel nom témoignerait que vous n'êtes pas raisonnable, mais plusieurs avis existent sur le sujet.

```
#
# Sauvegarder mes deux dossiers de musique
#
backup '/home/russ/oldies', music_files
backup '/home/russ/newies', music_files
to '/tmp/backup' interval 60
```

Le code précédent crée en amont une expression de recherche de fichiers qui est ensuite utilisée dans les deux instructions `backup`.

Améliorer PackRat

Malgré le fait que notre implémentation de PackRat est opérationnelle, elle est quelque peu limitée, car on ne peut spécifier qu'une seule configuration de sauvegarde à la fois. Pas de chance, l'implémentation actuelle ne permet pas d'utiliser deux ou trois dossiers de sauvegarde ou de copier certains fichiers avec un autre intervalle de temps. L'autre problème, c'est que PackRat n'est pas très propre : il repose en effet sur les méthodes du niveau supérieur `interval`, `to` et `backup`.

On peut résoudre ce problème en remaniant la syntaxe du fichier packrat.pr pour qu'un utilisateur crée et configure des instances multiples de `Backup` :

```
Backup.new do |b|
  b.backup '/home/russ/oldies', file_name('*.mp3') | file_name('*.wav')
  b.to '/tmp/backup'
  b.interval 60
end

Backup.new do |b|
  b.backup '/home/russ/newies', file_name('*.mp3') | file_name('*.wav')
  b.to '/tmp/backup'
  b.interval 60
end
```

Commençons par la classe `Backup` et voyons comment réaliser cette approche :

```
class Backup
  attr_accessor :backup_directory, :interval
  attr_reader :data_sources
  def initialize
    @data_sources = []
    @backup_directory = '/backup'
    @interval = 60
    yield(self) if block_given?
    PackRat.instance.register_backup(self)
  end

  def backup(dir, find_expression=All.new)
    @data_sources << DataSource.new(dir, find_expression)
  end
```

```
  def to(backup_directory)
    @backup_directory = backup_directory
  end

  def interval(minutes)
    @interval = minutes
  end

  def run
    while true
      this_backup_dir = Time.new.ctime.tr(" :","_")
      this_backup_path = File.join(backup_directory, this_backup_dir)
      @data_sources.each {|source| source.backup(this_backup_path)}
      sleep @interval*60
    end
  end
end
```

Puisque l'utilisateur est autorisé à créer des multiples instances, la classe `Backup` ne peut plus être un singleton. Nous avons déplacé les méthodes `backup`, `to` et `interval` à l'intérieur de la classe `Backup`. Les deux autres modifications apparaissent dans la méthode `initialize`. La méthode `initialize` de la classe `Backup` exécute `yield` en se passant elle-même comme paramètre unique. Cela permet aux utilisateurs de configurer l'instance de `Backup` dans un bloc de code passé à `new` :

```
Backup.new do |b|
  # Configurer une nouvelle instance de Backup
end
```

La dernière modification porte sur la méthode `initialize` de `Backup` et permet à la nouvelle version de s'enregistrer désormais dans la classe `PackRat` :

```
class PackRat
  include Singleton
  def initialize
    @backups = []
  end

  def register_backup(backup)
    @backups << backup
  end

  def run
    threads = []
    @backups.each do |backup|
      threads << Thread.new {backup.run}
    end
    threads.each {|t| t.join}
  end
end
eval(File.read('backup.pr'))
PackRat.instance.run
```

La classe `PackRat` maintient une liste d'instances de `Backup` et démarre chacune d'elles à l'appel de `run` dans un thread séparé.

User et abuser des DSL internes

Comme nous l'avons découvert, les DSL internes offrent une occasion unique de maximiser notre efficacité face à certains types de problèmes. Mais, comme tous les outils, cette méthode a ses limites. Aussi fluide que la syntaxe de Ruby puisse être, la capacité d'analyse syntaxique d'un DSL interne fondé sur Ruby n'est pas infinie. Par exemple, il est probablement impossible d'écrire en Ruby un DSL interne capable d'analyser la syntaxe d'un fragment de code HTML.

Les messages d'erreurs constituent un autre point non négligeable. Si vous n'êtes pas extrêmement prudent, les erreurs de programmes DSL peuvent produire des messages d'erreur assez étranges. Par exemple, que se passerait-il si un utilisateur malchanceux saisissait `x` au lieu de `b` dans le fichier `backup.pr` :

```
Backup.new do |b|
  b.backup '/home/russ/newies', name('*.mp3') | name('*.wav')
  b.to '/tmp/backup'
  x.interval 60
end
```

Il obtiendrait le message d'erreur suivant :

```
./ex6_multi_backup.rb:86: undefined local variable or method 'x' ...
```

Ce message est complètement incompréhensible pour un utilisateur qui ne connaît pas Ruby et qui essaie tout simplement de sauvegarder ses fichiers. Ce problème peut être atténué grâce à une gestion d'exceptions soigneusement élaborée. Néanmoins, les messages d'erreurs "cryptiques" représentent un écueil fréquent pour les DSL internes.

Enfin, si la sécurité est un point critique, évitez les DSL internes. Au fond, l'idée principale d'un DSL interne consiste à absorber dans votre programme du code arbitraire provenant de l'extérieur. Cette approche nécessite une confiance absolue.

Les DSL internes dans le monde réel

Rake, la réponse de Ruby à `ant` et `make`, constitue un des exemples les plus connus d'un DSL interne écrit en Ruby. La syntaxe de rake ressemble à celle de la deuxième version de PackRat, qui autorise des sauvegardes multiples.

L'utilitaire rake permet de spécifier des étapes qui constituent des tâches d'un processus d'installation. Les tâches peuvent être interdépendantes. Si la tâche B dépend de la

tâche A, rake exécuterait A avant B. Voici un exemple simple, le fichier rake suivant sert à sauvegarder mes dossiers de musique :

```
#
# Dossiers qui contiennent ma collection de musique
#
OldiesDir = '/home/russ/oldies'
NewiesDir = '/home/russ/newies'

#
# Dossier de sauvegarde
#
BackupDir = '/tmp/backup'

#
# Nom du dossier unique pour cette sauvegarde
#
timestamp=Time.new.to_s.tr(" :", "_")

#
# Les tâches rake
#
task :default => [:backup_oldies, :backup_newies]

task :backup_oldies do
  backup_dir = File.join(BackupDir, timestamp, OldiesDir)
  mkdir_p File.dirname(backup_dir)
  cp_r OldiesDir, backup_dir
end

task :backup_newies do
  backup_dir = File.join(BackupDir, timestamp, NewiesDir)
  mkdir_p File.dirname(backup_dir)
  cp_r NewiesDir, backup_dir
end
```

Ce fichier rake définit trois tâches. Les tâches `backup_oldies` et `backup_newies` effectuent exactement ce que suggère leur nom. La troisième tâche dépend des deux premières. Lorsque rake essaie d'exécuter la tâche `default`, il doit d'abord effectuer `backup_oldies` et `backup_newies`.

Rails est évidemment un autre exemple de définition de DSL. Contrairement à rake, Rails n'est pas un langage DSL interne pur, mais il est rempli de fonctionnalités inspirées par ce pattern. Parfois, on peut complètement oublier que l'on code en Ruby. Pour vous donner un exemple remarquable, ActiveRecord permet de spécifier des relations de classe d'une façon qui ressemble beaucoup à un DSL :

```
class Manager < ActiveRecord::Base
  belongs_to :department
  has_one :office
  has_many :committees
end
```

En conclusion

Le pattern Langage spécifique d'un domaine (ou Domain-Specific Language) est le premier pattern examiné dans ce livre qui ne fait pas partie des patterns répertoriés par le GoF. Mais ne soyez pas découragé car, lorsqu'on combine la syntaxe extrêmement flexible de Ruby avec la technique de DSL interne, on obtient un outil qui apporte énormément de puissance et de flexibilité sans pour autant nécessiter beaucoup de code. L'idée sous-jacente de ce pattern est assez simple : vous définissez votre DSL, qui s'inscrit dans les règles de syntaxe de Ruby ; ensuite, vous définissez une infrastructure nécessaire pour permettre l'exécution de programmes écrits dans votre langage DSL. La cerise sur le gâteau est apportée par le fait qu'un simple appel à la méthode `eval` permet d'exécuter votre programme DSL comme s'il s'agissait de code Ruby habituel.

17

Créer des objets personnalisés par méta-programmation

Au Chapitre 13, nous avons étudié deux patterns Factory proposés par le GoF. Ces deux patterns tentent de résoudre un des problèmes fondamentaux de la programmation orientée objet : comment obtenir un objet adapté à un problème ? Comment choisir le bon analyseur syntaxique pour les données à traiter ? Comment sélectionner l'adaptateur compatible avec la base de données disponible ? Quel objet de gestion de la sécurité sélectionner en fonction de la version des spécifications fournies ?

Les patterns Factory permettent justement de choisir la bonne classe qui se chargera de l'instanciation de l'objet. L'importance de choisir une classe est parfaitement justifiée dans des langages à typage statique. Lorsque le comportement d'un objet est complètement défini par sa classe et que ses classes ne sont pas modifiables au moment de l'exécution, sélectionner la bonne classe est le seul point critique.

Mais nous avons déjà appris que ces règles statiques ne sont pas applicables en Ruby. Ruby permet de modifier une classe existante, de changer le comportement d'un objet indépendamment de sa classe et même d'évaluer des chaînes de caractères comme du code Ruby au moment de l'exécution. Dans ce chapitre, nous découvrirons le pattern Méta-programmation, qui propose de profiter de ce comportement dynamique pour accéder aux objets requis. Avec ce pattern, on part du principe que les classes, les méthodes et le code à l'intérieur des méthodes sont simplement des composants du langage Ruby. Un bon moyen d'obtenir les objets requis serait donc de manipuler ces composants tout comme on manipule des entiers et des chaînes de caractères. Cette approche ne doit pas vous effrayer. Certes, la méta-programmation adopte une approche moins traditionnelle pour créer l'objet dont vous avez besoin mais, en réalité, c'est

un moyen de profiter de la flexibilité et de la nature dynamique de Ruby que j'évoque sans cesse dans ce livre.

En guise d'introduction à la méta-programmation[1], essayons encore une fois de créer des habitants dans notre simulateur environnemental.

Des objets sur mesure, méthode par méthode

Imaginez que nous sommes de retour dans la jungle du Chapitre 13 et que nous essayons de la peupler avec des plantes et des animaux. L'approche adoptée au Chapitre 13 consistait à recourir à une factory pour sélectionner les classes de flore et de faune adaptées. Et si l'on avait besoin une fois de plus de flexibilité ? Par exemple, au lieu de sélectionner un type d'organisme spécifique dans une liste de choix prédéfinis, on aimerait spécifier les propriétés de l'organisme et obtenir un objet construit sur mesure ? On pourrait par exemple avoir une méthode de création des plantes qui accepterait des paramètres décrivant le type de plante requis. Cela nous permettrait de construire l'objet dynamiquement au lieu de choisir explicitement une classe adaptée :

```
def new_plant(stem_type, leaf_type)
  plant = Object.new
  if stem_type == :fleshy
    def plant.stem
      'fleshy'
    end
  else
    def plant.stem
      'woody'
    end
  end
  if leaf_type == :broad
    def plant.leaf
      'broad'
    end
  else
    def plant.leaf
      'needle'
    end
  end
  plant
end
```

1. Il faut se rendre compte que la communauté Ruby est assez partagée en ce qui concerne la définition exacte de ce qu'est la méta-programmation. Dans ce chapitre, j'ai tenté d'englober le maximum de concepts de la méta-programmation sans me préoccuper de la définition précise du terme.

Le code précédent crée une instance classique d'`Object` et modifie ensuite cet objet selon les spécifications fournies par le client. La méthode `new_plant` équipe l'objet des méthodes `leaf` et `stem` spécifiques en fonction des options reçues. L'objet final est plus ou moins unique : la plupart de ses fonctionnalités proviennent non pas de sa classe (ce n'est qu'un `Object`) mais plutôt de ces méthodes singleton. L'objet retourné par `new_plant` est effectivement fait sur mesure.

L'utilisation de la méthode `new_plant` est très simple. Il suffit de spécifier quel type de plante il vous faut :

```
plant1 = new_plant(:fleshy, :broad)
plant2 = new_plant(:woody, :needle)
puts "Plant 1's stem: #{plant1.stem} leaf: #{plant1.leaf}"
puts "Plant 2's stem: #{plant2.stem} leaf: #{plant2.leaf}"
```

Voici le résultat :

```
Plant 1's stem: fleshy leaf: broad
Plant 2's stem: woody leaf: needle
```

Évidemment, il n'existe pas de règle qui vous oblige à commencer votre personnalisation sur une instance simple d'`Object`. Dans la réalité, on serait susceptible d'instancier une classe qui fournit un certain niveau de fonctionnalité et de modifier les méthodes de cette instance.

La technique de personnalisation est particulièrement utile lorsque vous avez un ensemble de fonctionnalités orthogonales qu'il faut rassembler dans un objet unique. La construction des objets sur mesure permet d'éviter de définir un tas de classes avec des noms comme `WoodyStemmedNeedleLeafFloweringPlant` et `VinyStemmedBroadLeafNonfloweringPlant`.

Des objets sur mesure, module par module

Si vous ne souhaitez pas créer vos objets méthode par méthode, vous pouvez toujours les personnaliser avec des modules. Supposons que vous ayez des modules séparés pour des carnivores et des herbivores :

```
module Carnivore
  def diet
    'meat'
  end

  def teeth
    'sharp'
  end
end
```

```
module Herbivore
  def diet
    'plant'
  end

  def teeth
    'flat'
  end
end
```

On peut alors imaginer une autre collection de modules qui représentent des animaux actifs en journée, comme des gens (ou la plupart des gens), et ceux qui rôdent pendant la nuit :

```
module Nocturnal
  def sleep_time
    'day'
  end

  def awake_time
    'night'
  end
end

module Diurnal
  def sleep_time
    'night'
  end

  def awake_time
    'day'
  end
end
```

Puisque les méthodes sont organisées dans des modules, le code nécessaire pour créer de nouveaux objets est plus concis :

```
def new_animal(diet, awake)
  animal = Object.new
  if diet == :meat
    animal.extend(Carnivore)
  else
    animal.extend(Herbivore)
  end
  if awake == :day
    animal.extend(Diurnal)
  else
    animal.extend(Nocturnal)
  end
  animal
end
```

La méthode `extend` appelée dans ce code a exactement le même effet que l'inclusion d'un module normal, `extend` est simplement plus commode pour modifier des objets à la volée.

Quelle que soit la technique adoptée, modification par ajout de méthodes ou de modules, le résultat aboutit à la création d'un objet personnalisé, construit sur mesure selon vos spécifications.

Ajouter de nouvelles méthodes

Imaginons que vous ayez reçu une nouvelle demande d'évolution pour votre simulateur d'habitats : les clients veulent modéliser des populations différentes de plantes et d'animaux. Ils aimeraient par exemple pouvoir grouper tous les êtres vivants qui habitent une zone précise, grouper tous les tigres et les arbres qui partagent la même section de la jungle, ou grouper des jungles situées côte à côte. Et ce serait bien de pouvoir ajouter un code pour suivre la classification biologique de tous ces êtres vivants. Les clients souhaitent savoir qu'un tigre fait partie du genre Panthera, de la famille Felidae, et ainsi de suite jusqu'au règne Animal.

Au premier abord, nous sommes face à deux problèmes de programmation distincts : d'une part organiser les organismes par situation géographique et d'autre part les organiser par classification biologique. Les deux problèmes semblent être similaires et ressemblent fort au pattern Composite. Mais il faudrait s'asseoir et écrire du code pour gérer les populations ainsi que du code pour gérer la classification, n'est-ce pas ? Peut-être pas. Il est peut-être possible d'extraire les aspects communs et d'implémenter une fonctionnalité qui résout les deux problèmes à la fois.

Parfois, le meilleur moyen de s'attaquer à une tâche pareille consiste à imaginer le résultat final et en déduire l'implémentation. Idéalement, on aimerait déclarer que l'instance d'une classe donnée – par exemple `Frog` ou `Tiger`[1] – fait partie d'une population géographique, ou d'une classification biologique, ou des deux, comme ceci :

```
class Tiger < CompositeBase
  member_of(:population)
  member_of(:classification)
  # Beaucoup de code omis...
end

class Tree < CompositeBase
  member_of(:population)
  member_of(:classification)
  # Beaucoup de code omis...
end
```

1. Dans cette section, je retourne vers l'implémentation traditionnelle de `Tiger` et `Tree` fondée sur des classes. Cela ne signifie pas que les différentes techniques de méta-programmation soient incompatibles. Mais une tentative d'expliquer tout à la fois ne serait pas compatible avec la notion d'explication compréhensible.

On essaie d'exprimer par ce code le fait que les classes `Tiger` et `Tree` sont des feuilles de deux composites différents. L'un conserve un suivi des populations par répartition géographique et l'autre modélise les classifications biologiques.

On doit également déclarer que les classes qui représentent les espèces et les populations géographiques sont des composites :

```
class Jungle < CompositeBase
  composite_of(:population)
  # Beaucoup de code omis...
end

class Species < CompositeBase
  composite_of(:classification)
  # Beaucoup de code omis...
end
```

Idéalement, l'utilisation de nos nouvelles classes `Tiger`, `Tree`, `Jungle` et `Species` doit être extrêmement simple. On aimerait par exemple pouvoir créer un tigre et ensuite l'ajouter à une population géographique donnée :

```
tony_tiger = Tiger.new('tony')
se_jungle = Jungle.new('southeastern jungle tigers')
se_jungle.add_sub_population(tony_tiger)
```

Une fois cette opération accomplie, on doit avoir la possibilité de retrouver la population parent de notre tigre :

```
tony_tiger.parent_population  # Le résultat doit être 'southeastern
                              # jungle'
```

Enfin, la même approche doit fonctionner pour des classifications biologiques :

```
species = Species.new('P. tigris')
species.add_sub_classification(tony_tiger)
tony_tiger.parent_classification # Le résultat doit être 'P. tigris'
```

Ci-après, on trouve la classe `CompositeBase`, qui implémente toute cette magie :

```
class CompositeBase
  attr_reader :name
  def initialize(name)
    @name = name
  end

  def self.member_of(composite_name)
    code = %Q{
      attr_accessor :parent_#{composite_name}
    }
    class_eval(code)
  end
```

```
  def self.composite_of(composite_name)
    member_of composite_name
    code = %Q{

    def sub_# {composite_name}s
      @sub_#{composite_name}s = [] unless @sub_#{composite_name}s
      @sub_#{composite_name}s
    end

    def add_sub_#{composite_name}(child)
      return if sub_#{composite_name}s.include?(child)
      sub_#{composite_name}s << child
      child.parent_#{composite_name} = self
    end

    def delete_sub_#{composite_name}(child)
      return unless sub_#{composite_name}s.include?(child)
      sub_#{composite_name}s.delete(child)
      child.parent_#{composite_name} = nil
    end
    }
    class_eval(code)
  end
end
```

Analysons cette classe étape par étape. Le début de la classe `CompositeBase` est assez conventionnel : on définit une simple variable d'instance et la méthode `initialize` pour y affecter une valeur. Les choses deviennent intéressantes dans la deuxième méthode, la méthode de classe `member_of` :

```
def self.member_of(composite_name)
  code = %Q{
    attr_accessor :parent_#{composite_name}
  }
  class_eval(code)
end
```

La méthode accepte le nom du composite correspondant et construit un fragment de code Ruby en se fondant sur ce nom. Si vous appelez `member_of` et passez `:population` en argument (comme la classe `Tiger`), la méthode `member_of` génère la chaîne de caractères suivante :

```
attr_accessor :parent_population
```

Ensuite, la méthode `member_of` fait appel à la méthode `class_eval` pour évaluer la chaîne en tant que code Ruby. La méthode `class_eval` ressemble à la méthode `eval` que nous connaissons déjà, à la seule différence que `class_eval` évalue une chaîne dans le contexte de la classe au lieu du contexte courant[1]. Vous avez probablement

1. La méthode `class_eval` est également connue sous le nom de `module_eval`.

deviné que cela a pour effet d'ajouter à la classe un accesseur et un mutateur de la variable d'instance `parent_population`. C'est précisément la modification qui est nécessaire pour transformer la classe en un membre (une feuille, pour être plus précis) d'un composite.

La méthode suivante de la classe `CompositeBase`, nommée `composite_of`, suit le même principe. Elle ajoute des méthodes qui correspondent à un objet composite. Si vous appelez la méthode de classe `composite_of` d'une de vos classes, elle acquiert trois nouvelles méthodes : une méthode pour ajouter un élément au composite, une méthode pour supprimer un élément et une méthode pour retourner un tableau qui contient la totalité des éléments. Puisque nous générons toutes ces méthodes par la création d'une chaîne de caractères et son évaluation avec `class_eval`, il est facile d'insérer dans les noms des méthodes le nom du composite. Les méthodes créées par l'appel `member_of (:population)` sont donc `add_sub_population`, `delete_sub_population` et `sub_populations`.

Le point clé à retenir à propos de cet exemple est que les sous-classes de `CompositeBase` n'héritent pas automatiquement du comportement du pattern Composite. Elles n'héritent que des méthodes de classe `member_of` et `composite_of`, qui ajoutent les méthodes composites à la sous-classe lorsqu'elles sont invoquées.

L'objet vu de l'intérieur

La technique d'ajout de fonctionnalités telle que nous l'avons utilisée dans la classe `CompositeBase` soulève une question : comment peut-on savoir si un objet donné fait partie d'un composite ? Plus généralement, lorsqu'on injecte des fonctionnalités dans des classes à la volée à l'aide de la méta-programmation, comment peut-on savoir quelles fonctionnalités sont disponibles dans une instance donnée ?

Eh bien, il suffit simplement de se renseigner auprès de cette instance ! Des objets Ruby fournissent une palette très complète de méthodes dites "de réflexion" qui renvoient différentes informations sur un objet, par exemple quelles sont ses méthodes ou ses variables d'instance. Pour découvrir si un objet fait partie d'un composite comme `CompositeBase`, on peut inspecter la liste de ses méthodes publiques :

```
def member_of_composite?(object, composite_name)
  public_methods = object.public_methods
  public_methods.include?("parent_#{composite_name}")
end
```

On peut aussi avoir recours à la méthode `respond_to?` :

```
def member_of_composite?(object, composite_name)
  object.respond_to?("parent_#{composite_name}")
end
```

Des fonctions de réflexion telles que `public_methods` et `respond_to?` sont effectivement très pratiques, mais on peut même les qualifier de totalement indispensables lorsqu'on plonge dans le monde de la méta-programmation car vos objets dépendent alors beaucoup plus de l'historique des modifications subies à l'exécution que de leurs classes de base.

User et abuser de la méta-programmation

Chaque pattern doit être employé lorsqu'il est adapté aux circonstances. C'est plus vrai que jamais pour la méta-programmation, qui est un outil extrêmement puissant. Avec la méta-programmation, le code de l'application lui-même subit des changements au moment de l'exécution. Plus vous utilisez la méta-programmation, moins le programme exécuté ressemble au programme qui se trouve dans les fichiers source. C'est évidemment le but, mais c'est aussi le danger. Déboguer du code ordinaire est assez difficile, et il est encore plus compliqué de corriger des erreurs dans un programme éphémère généré par la méta-programmation. Alors que les tests unitaires sont importants pour des programmes classiques, ils deviennent absolument vitaux dans des systèmes qui utilisent la méta-programmation intensivement.

Le danger principal de ce pattern est l'interaction inattendue entre plusieurs fonctions. Imaginez le chaos dans notre exemple des habitats si la méthode `parent_classification` était déjà définie dans la classe `Species` lorsqu'elle appelle `composite_of(:classification)` :

```
class Species < CompositeBase
  # Cette méthode est sur le point de disparaître !
  def parent_classification
    # ...
  end
  # Voici le code qui provoque sa destruction...
  composite_of(:classification)
end
```

Parfois, il est possible de prévenir ce genre de carnage en ajoutant des garde-fous à votre méta-code :

```
class CompositeBase
  # ...
```

```
    def self.member_of(composite_name)
      attr_name = "parent_#{composite_name}"
      raise 'Method redefinition' if instance_methods.include?(attr_name)
      code = %Q{
        attr_accessor :#{attr_name}
      }
      class_eval(code)
    end
    # ...
  end
```

Cette version de `CompositeBase` déclenche une exception si la méthode `parent_<composite_name>` existe déjà. L'approche n'est pas idéale, mais elle est probablement meilleure que passer sous silence l'écrasement d'une méthode existante.

La méta-programmation dans le monde réel

Chercher des exemples de méta-programmation dans la base de code Ruby est comme chercher des vêtements sales dans la chambre de mon fils : il y en a partout. Regardez par exemple l'omniprésent `attr_accessor` et ses amis `attr_reader` et `attr_writer`. Souvenez-vous du Chapitre 2, où nous avons découvert que toutes les variables d'instance Ruby sont privées et nécessitent des accesseurs et mutateurs pour ouvrir l'accès au monde extérieur :

```
class BankAccount
  def initialize(opening_balance)
    @balance = opening_balance
  end

  def balance
    @balance
  end

  def balance=(new_balance)
    @balance = new_balance
  end
end
```

Au Chapitre 2, nous avons également appris la bonne nouvelle que nous ne sommes pas obligés d'écrire toutes ces méthodes de base. Selon nos besoins, nous pouvons simplement insérer `attr_reader`, `addr_writer` ou la combinaison des deux `attr_accessor` :

```
class BankAccount
  attr_accessor :balance
  def initialize(opening_balance)
    @balance = opening_balance
  end
end
```

Il s'avère qu'`attr_accessor` et ses copains `reader` et `writer` ne sont pas des mots clés réservés du langage Ruby. Ce sont juste des méthodes de classe ordinaires[1], tout comme les méthodes `member_of` et `composite_of` élaborées dans ce chapitre.

Il est en effet assez simple d'écrire notre propre version d'`attr_reader`. Étant donné que le nom "attr_reader" est déjà occupé, appelons notre méthode `readable_attribute` :

```
class Object
  def self.readable_attribute(name)
    code = %Q{
      def #{name}
        @#{name}
      end
    }
    class_eval(code)
  end
end
```

Une fois la méthode `readable_attribute` définie, on peut l'utiliser de la même manière qu'`attr_reader` :

```
class BankAccount
  readable_attribute :balance
  def initialize(balance)
    @balance = balance
  end
end
```

Le module `Forwardable` est un autre bon exemple auquel nous avons fait appel pour construire des décorateurs. La création fastidieuse des méthodes de délégation devient automatique avec le module `Forwardable`. Par exemple, si l'on avait une classe `Car` avec une classe `Engine` séparée, on pourrait écrire ceci :

```
class Engine
  def start_engine
    # Démarrer le moteur
  end

  def stop_engine
    # Arrêter le moteur
  end
end
```

1. La méthode `attr_accessor` et ses amis résident dans le module `Module` inclus dans la classe `Object`. Si vous partez à la recherche des méthodes `attr_accessor`, `attr_reader` et `attr_writer` dans le code Ruby, vous risquez d'être déçu. Pour des raisons de performance – et uniquement pour ces raisons –, ces méthodes sont écrites en C.

```
class Car
  extend Forwardable
  def_delegators :@engine, :start_engine, :stop_engine
  def initialize
    @engine = Engine.new
  end
end
```

La ligne qui commence avec `def_delegators` crée deux méthodes : `start_engine` et `stop_engine`. Chacune d'elles délègue à l'objet référencé par `@engine`. Le module `Forwardable` crée ces méthodes à l'aide de la même technique fondée sur `class_eval` que celle étudiée dans ce chapitre.

Il ne faut pas oublier Rails. Le volume de méta-programmation dans Rails est si énorme qu'il est difficile de choisir par où commencer. La façon de définir les relations entre les tables dans ActiveRecord constitue probablement l'exemple le plus remarquable. ActiveRecord fournit une classe pour chaque table de la base de données. Si l'on modélise un habitat naturel avec ActiveRecord, on pourrait par exemple avoir une table – et une classe – pour les animaux. On pourrait également modéliser la description complète de chaque animal dans une table séparée. Il est évident que les tables des animaux et des descriptions auront une relation un-à-un. ActiveRecord nous permet d'exprimer cette relation de manière très élégante :

```
class Animal < ActiveRecord::Base
  has_one :description
end

class Description < ActiveRecord::Base
  belongs_to :animal
end
```

On peut exprimer de manière similaire toutes les relations courantes entre des tables dans une base de données. Par exemple, chaque espèce pourrait inclure beaucoup d'animaux :

```
class Species < ActiveRecord::Base
  has_many : animals
end
```

Mais chaque animal appartient à une espèce unique :

```
class Animal < ActiveRecord::Base
  has_one :description
  belongs_to :species
end
```

Toutes ces déclarations ont pour conséquence l'injection de code dans les classes `Animal`, `Description` et `Species`. Ce code permet d'établir entre les classes les relations définies dans la base de données. Une fois la relation spécifiée, on peut demander à une instance d'`Animal` de retourner sa description correspondante avec un simple appel `animal.description`, ou bien nous pouvons récupérer tous les animaux qui font partie d'une espèce donnée à l'aide de `species.animals`. C'est grâce à la méta-programmation qu'ActiveRecord est capable de faire tout cela pour nous.

En conclusion

Dans ce chapitre, nous avons passé en revue quelques principes fondateurs de la méta-programmation et, notamment, qu'il est possible d'injecter le code nécessaire par programmation au moment de l'exécution au lieu de le saisir au clavier. Les fonctionnalités dynamiques de Ruby nous permettent de partir d'un objet simple et d'y ajouter des méthodes ou même des modules entiers remplis de méthodes. Des méthodes complètement nouvelles peuvent être générées au moment de l'exécution à l'aide de `class_eval`.

Nous avons aussi tiré parti des capacités de réflexion de Ruby, qui permettent à un programme d'examiner sa propre structure et de découvrir ce qu'il est déjà capable de faire avant de lui apporter des modifications.

Dans le monde réel, la méta-programmation est un des éléments de base pour les DSL, les langages spécifiques d'un domaine que nous avons découvert au Chapitre 16. Malgré le fait qu'il est possible de développer un DSL avec peu ou pas de méta-programmation – comme nous l'avons fait au Chapitre 16 –, la méta-programmation est souvent l'ingrédient primordial dans le développement des DSL à la fois puissants et faciles d'utilisation.

Au chapitre suivant, nous terminerons l'analyse des design patterns en Ruby avec un autre pattern qui s'inscrit bien dans la philosophie de la méta-programmation. Il s'agit du principe nommé "Convention plutôt que configuration".

18

Convention plutôt que configuration

Le dernier pattern que nous examinons dans ce livre se nomme "Convention plutôt que configuration". Le livre du GoF *Design Patterns* n'est pas à l'origine de ce pattern, car il vient directement du framework Rails. On peut affirmer sans ambage que le pattern Convention plutôt que configuration fait partie des clés du succès de Rails. C'est un pattern ambitieux d'une très grande portée, ce qui le distingue des autres patterns présentés dans ce livre. Alors que les autres patterns interviennent sur une plus petite échelle et sont principalement concernés par des problèmes d'organisation d'un certain nombre de classes interdépendantes, Convention plutôt que configuration se concentre sur l'organisation des applications et des frameworks entiers. Comment structurer une application ou un framework de façon à permettre aux autres ingénieurs d'ajouter facilement du code au fur et à mesure de l'évolution du programme ? Lorsque nous construisons des systèmes de plus en plus ambitieux, le problème de leur configuration devient plus que jamais d'actualité.

La réaction du monde du logiciel face au problème de l'extensibilité me rappelle le dilemme que je rencontrais tous les soirs lorsque j'étais au collège. Je ne pouvais jamais décider quand il fallait faire mes devoirs. Il y avait des jours où je revenais de l'école, j'ouvrais mes livres et je faisais mes devoirs. Il n'y a rien de mieux que le sentiment du travail accompli. Mais, par ailleurs, il n'y a rien de mieux que revenir à la maison, jeter ses livres sur la table, sauter sur son vélo et partir à l'aventure. À mon retour, les livres étaient toujours là, bien entendu, et tôt ou tard je devais faire mes devoirs quand même. En fin de compte, j'avais fini par trouver un compromis : je m'occupais du français tant détesté et des ennuyeuses sciences sociales directement après l'école et je remettais les mathématiques, que j'adorais, à plus tard.

L'ingénierie logicielle a suivi un parcours assez semblable en ce qui concerne les capacités d'extension des systèmes d'information. Au fil de leur carrière de nombreux programmeurs ont travaillé avec des systèmes, fiers de leurs propres limitations. Ces programmes ne comprenaient qu'un protocole unique, ou bien exigeaient un schéma bien précis de la base de données, ou encore ils imposaient sur les malheureux utilisateurs une interface totalement rigide.

La réaction à toutes les difficultés provoquées par cette rigidité fut de rendre les systèmes extensibles lors de la phase de configuration. En effet, en déportant les décisions importantes dans un fichier de configuration on pourrait alors accéder à l'Utopie totale : la définition d'un système idéal uniquement par l'effet de la configuration. Malheureusement, on est passé à côté de l'Utopie pour arriver à nouveau dans un désert où le code est maintenant totalement dépendant de la configuration. Aujourd'hui, nous sommes envahis par des frameworks sensibles à la moindre modification de la configuration et des applications qui vivent avec la peur que toute approximation injustifiée devra faire l'objet d'un patch en urgence.

Les servlets Java sont un parfait exemple de ce genre de configuration excessive. Les servlets sont des composants critiques de quasiment toutes les applications Web écrites en Java. Un servlet est une petite classe Java élégante capable de traiter des requêtes HTTP qui arrivent à partir d'une ou de plusieurs URL. Mais, malheureusement, pour écrire un servlet il ne suffit pas d'écrire une classe Java qui étend `javax.servlet.HttpServlet`. Vous devez également configurer votre classe à l'aide du fichier de configuration web.xml. La forme la plus basique de web.xml permet d'associer la classe servlet avec un nom, puis d'associer ce nom avec une ou plusieurs URL.

Pourtant, les applications réelles ont rarement besoin de toute cette flexibilité. Dans la plupart des cas – pas toujours, mais très souvent –, on a tendance à utiliser le nom de la classe en tant que nom de servlet. Dans la plupart des cas – pas toujours, mais très souvent –, on associe ce nom à une URL dérivée de ce nom et par conséquent au nom de la classe. On pourrait choisir n'importe quel nom mais il est bien mieux d'utiliser celui qui nous rappelle le nom de notre classe. De la même manière, peu importe l'URL qui sert de porte d'entrée à notre servlet, mais c'est tout de même bien mieux si lui aussi utilise le même nom que notre classe. Les programmeurs (les bons programmeurs, tout au moins) attachent beaucoup de valeur à la simplicité. La solution simple dans cette situation serait d'éliminer complètement toute cette flexibilité. Si vous n'avez aucune utilité de la flexibilité qui vous est offerte, elle devient un danger : tous ces noms et associations représentent autant de moyens supplémentaires de se tirer une balle dans le pied.

La motivation principale du pattern Convention plutôt que configuration est précisément d'alléger le fardeau de la configuration. Le but est de préserver l'extensibilité essentielle de nos applications et de nos frameworks tout en supprimant la configuration excessive. Dans ce chapitre, nous commencerons par étudier les principes du pattern Convention plutôt que configuration. Ensuite, nous développerons un système de messages hypothétiques pour comprendre comment concevoir des logiciels qui trouvent un compromis judicieux, comme j'avais su le faire pour mes devoirs scolaires.

Une bonne interface utilisateur... pour les développeurs

L'écriture de logiciels à la fois flexibles et simples à utiliser est un problème courant surtout si vous développez des interfaces graphiques. Les programmeurs d'interfaces graphiques ont élaboré plusieurs règles de conception pour créer des interfaces conviviales :

- *Essayez d'anticiper les besoins des utilisateurs*. Dans une interface bien conçue, les tâches les plus courantes ne doivent requérir quasiment aucun effort, et le cas le plus courant doit être le choix par défaut. Les tâches moins fréquentes ou plus complexes doivent rester faisables avec un peu plus d'effort.
- *N'obligez pas les utilisateurs à se répéter.* Est-ce qu'il vous est arrivé de vouloir donner un coup de pied dans l'écran lorsque l'application vous demande pour la troisième fois : "Êtes-vous sûr de vouloir faire cette action ?"
- *Fournissez des modèles.* Présentez à l'utilisateur un modèle à suivre pour l'aider à bâtir quelque chose. N'imposez pas à votre utilisateur l'angoisse de la feuille blanche : s'il souhaite écrire un CV, fournissez un modèle pour lui donner des idées.

Le pattern Convention plutôt que configuration se concentre sur l'application de ces mêmes principes transposés dans la conception d'applications et des API de frameworks. Il n'y a aucune raison pour que ces bonnes techniques ne profitent qu'à l'utilisateur final. Les ingénieurs qui essaient de paramétrer votre application ou de faire appel à votre API sont aussi des utilisateurs et ont besoin d'aide. Pourquoi ne pas offrir une interface conviviale à tous vos utilisateurs ?

Anticiper les besoins

Que ce soit un client de courrier électronique ou une API, il y a une caractéristique qui aide à distinguer les bonnes interfaces des mauvaises : si l'utilisateur exécute une action très fréquemment, cela doit être l'option par défaut. Inversement, si l'action n'est effectuée que rarement, il est tout à fait acceptable qu'elle soit plus difficile d'accès. C'est la

raison pour laquelle il suffit d'appuyer sur une touche pour passer d'un message électronique à l'autre, mais il faut naviguer dans plusieurs menus pour configurer un serveur d'e-mail.

Très souvent des API sont construites en partant du principe que la fréquence d'utilisation des différentes fonctions sera identique. C'est cette supposition qui a amené à la configuration des servlets Java : peu importe si l'on crée une servlet ordinaire qui ne répond qu'à une seule URL ou une servlet complexe et multifonction liée à plusieurs URL, le temps de développement et de configuration de la tâche la plus simple est le même que celui de la tâche la plus complexe ou la moins fréquente. Une interface de programmation plus conviviale rendrait la tâche courante la plus simple possible mais demanderait plus de travail pour accomplir la tâche la plus complexe.

Ne le dire qu'une seule fois

Pour rendre fous vos utilisateurs, il suffit de les obliger à se répéter. C'est une leçon bien connue dans le monde des interfaces graphiques, mais à l'évidence elle n'a pas été apprise dans le monde des programmeurs d'API. Comment éviter à nos développeurs des répétitions inutiles ? Il suffit de leur donner le moyen d'exprimer leurs souhaits une fois et de ne plus leur redemander. Des ingénieurs ont tendance à naturellement adopter des conventions : ils suivent des règles de nommage de fichiers, ils regroupent des fichiers source liés entre eux dans les mêmes dossiers et ils nomment des méthodes selon des patrons réguliers.

Le principe du pattern Convention plutôt que configuration consiste précisément à définir une convention probablement déjà appliquée par des ingénieurs raisonnables : placer tous les adaptateurs dans un dossier précis ou nommer toutes les méthodes de contrôle d'accès d'une certaine façon. Pour établir une bonne convention, tout comme pour concevoir une bonne interface graphique, il faut se mettre à la place de ses utilisateurs. Essayez pour cela d'anticiper leur comportement : comment vont-ils appeler certaines fonctions et quelle serait pour eux la façon la plus naturelle de les organiser ; ensuite, mettez en place votre convention en vous fondant sur ces hypothèses. Lorsque votre convention est prête, tâchez de maximiser son utilité : lorsqu'un ingénieur nomme une classe et la place dans un répertoire donné, il exprime quelque chose. Soyez attentif et ne l'obligez pas à se répéter.

Fournir un modèle

Un autre moyen de faciliter l'utilisation de votre système consiste à fournir à vos utilisateurs un modèle ou un exemple. Les éditeurs de texte modernes, par exemple, ne vous

laissent plus seul face à la page blanche. Lorsque vous créez un document, le programme vous demande si c'est un CV, une lettre ou un discours présidentiel et, si votre document fait partie des dizaines de modèles connus, l'éditeur de texte vous proposera un contenu et une mise en page adaptés.

Vous pouvez rendre le même service aux programmeurs qui essaient d'étendre votre système : proposez des exemples, des modèles et des fragments de code pour les aider à bien démarrer. Si un dessin vaut mieux qu'un long discours, alors, un ou deux bons exemples valent certainement au moins deux cents pages de documentation.

Une passerelle de messages

Voyons comment ces nobles idéaux s'appliquent au code réel. Imaginons qu'on nous ait demandé de développer une passerelle de messages. Notre code doit être à même de recevoir des messages et de les transférer vers leurs destinations finales. Les messages ressemblent à ceci :

```
require 'uri'
class Message
  attr_accessor :from, :to, :body
  def initialize(from, to, body)
    @from = from
    @to = URI.parse(to)
    @body = body
  end
end
```

Le champ `from` est une simple chaîne de caractères qui contient l'information sur l'expéditeur du message, quelque chose comme `'russ.olsen'`. Le champ `to` est un URI qui indique la destination du message. Le champ `body` est une chaîne qui correspond au corps du message. La classe `Message` emploie la classe `URI`, qui fait partie de la distribution standard de Ruby, pour convertir des chaînes en objets URI utilisables. Les URI entrants de notre passerelle sont de trois types : on peut être amené à envoyer un message soit par courrier électronique :

```
smtp://fred@russolsen.com
```

Soit par une requête HTTP Post :

```
http://russolsen.com/some/place
```

Soit l'écrire dans un fichier :

```
file:///home/messages/message84.txt
```

L'exigence clé pour notre passerelle de messages est la possibilité d'ajouter de nouveaux protocoles facilement. Par exemple, si l'on devait envoyer des messages par FTP, il faudrait pouvoir adapter la passerelle très simplement pour gérer les nouvelles destinations.

Après avoir consulté votre livre préféré sur les design patterns[1], vous vous rendez compte que pour traiter les différentes destinations de ces messages il vous faut un adaptateur. Trois adaptateurs, pour être plus précis : un adaptateur par protocole. Dans ce cas, l'interface de l'adaptateur est très simple, il ne consiste qu'en une seule méthode : `send_message(message)`. Voici l'adaptateur qui transfère des messages en tant que courrier électronique :

```
require 'net/smtp'
class SmtpAdapter
  MailServerHost = 'localhost'
  MailServerPort = 25
  def send_message(message)
    from_address = message.from
    to_address = message.to.user + '@' + message.to.host
    email_text = "From: #{from_address}\n"
    email_text += "To: #{to_address}\n"
    email_text += "Subject: Forwarded message\n"
    email_text += "\n"
    email_text += message.body
    Net::SMTP.start(MailServerHost, MailServerPort) do |smtp|
      smtp.send_message(email_text, from_address, to_address)
    end
  end
end
```

Voici l'adaptateur qui envoie des messages par HTTP :

```
require 'net/http'
class HttpAdapter
  def send_message(message)
    Net::HTTP.start(message.to.host, message.to.port) do |http|
      http.post(message.to.path, message.body)
    end
  end
end
```

Enfin, voici l'adaptateur qui "envoie" des messages en les copiant dans un fichier :

```
class FileAdapter
  def send_message(message)
    #
    # Récupérer le chemin de l'URL
```

1. Celui-ci, bien évidemment !

```
      # et effacer le premier '/'
      #
      to_path = message.to.path
      to_path.slice!(0)
      File.open(to_path, 'w') do |f|
        f.write(message.body)
      end
    end
  end
```

Sélectionner un adaptateur

Le problème suivant consiste à déterminer la classe adaptateur appropriée selon le message reçu. Une des solutions consiste à coder en dur la logique de sélection d'un adaptateur :

```
def adapter_for(message)
  protocol = message.to.scheme
  return FileAdapter.new if protocol == 'file'
  return HttpAdapter.new if protocol == 'http'
  return SmtpAdapter.new if protocol == 'smtp'
  nil
end
```

Toutefois, cette solution présente un problème : la personne qui ajoutera un nouveau protocole de livraison, et par conséquent un nouvel adaptateur, sera obligée de plonger dans le code de la méthode `adapter_for` pour y ajouter son nouvel adaptateur. Obliger quelqu'un à modifier le code existant ne rend assurément pas notre application "facilement extensible". On peut probablement mieux faire. On pourrait, par exemple, avoir un fichier de configuration pour définir la correspondance entre les protocoles et les noms des adaptateurs, comme ceci :

```
smtp: SmtpAdapter
file: FileAdapter
http: HttpAdapter
```

Cette solution pourrait fonctionner, mais ce fichier de configuration signifierait en réalité que nous avons opté pour un autre type de codage en dur. Dans les deux cas, la personne qui ajoute un nouvel adaptateur doit effectuer une manipulation supplémentaire afin que le système reconnaisse ce nouvel adaptateur.

Cela nous amène à la question de fond : pourquoi cela ne suffit-il pas d'écrire la nouvelle classe adaptateur pour qu'elle soit immédiatement prise en compte ? Si l'on demande aux auteurs de nouveaux adaptateurs d'adhérer à une convention judicieuse, le rajout d'un adaptateur peut se réduire à la création de la classe. Voici la convention magique : nommez votre classe adaptateur `<protocol>Adapter`.

Selon cette convention, un nouvel adaptateur capable d'envoyer des fichiers *via* FTP s'appellerait `FtpAdapter`. Si tous les adaptateurs suivaient cette convention, le système pourrait sélectionner la classe adaptateur en fonction de son nom[1] :

```
def adapter_for(message)
  protocol = message.to.scheme.downcase
  adapter_name = "#{protocol.capitalize}Adapter"
  adapter_class = self.class.const_get(adapter_name)
  adapter_class.new
end
```

La méthode `adapter_for` extrait le protocole cible du message, puis elle transforme le nom tel que `"http"` en `"HttpAdapter"` avec quelques manipulations de chaîne de caractères. Ensuite, il suffit d'appeler `const_get` pour récupérer la classe qui porte le même nom. Cette approche nous permet d'éviter l'utilisation d'un fichier de configuration : pour ajouter un nouvel adaptateur il suffit de créer la classe adaptateur en la baptisant correctement.

Charger des classes

Enfin presque... Il nous reste tout de même à gérer le chargement des adaptateurs dans l'interpréteur Ruby. Dans notre code nous devons inclure les fichiers qui contiennent les classes adaptateurs à l'aide de l'instruction `require` :

```
require 'file_adapter'
require 'http_adapter'
require 'smtp_adapter'
```

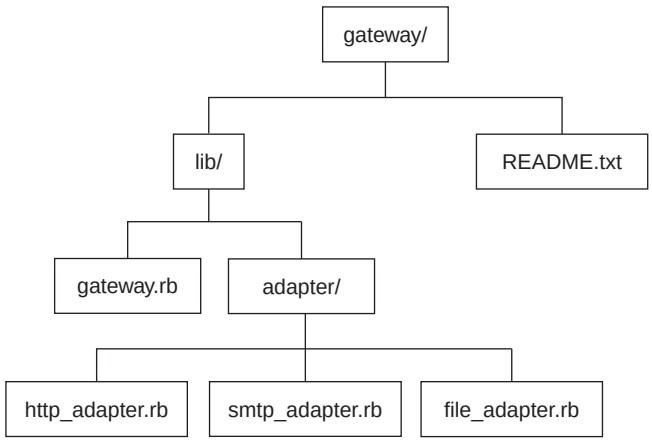
On pourrait placer toutes ces instructions `require` pour chacun des adaptateurs dans un fichier et prévenir les auteurs des adaptateurs de compléter ce fichier lorsqu'un nouvel adaptateur arrive. Mais, une fois de plus, cela signifie que l'on oblige le programmeur à se répéter : il doit non seulement créer l'adaptateur, mais aussi nous le confirmer en ajoutant une ligne dans le fichier des instructions `require`. On peut sûrement trouver une solution plus élégante.

Commençons par nous concentrer sur la structure des dossiers. D'habitude, on ne prête pas trop attention à l'emplacement physique des fichiers et dossiers où réside notre code source, alors qu'il est possible d'établir des conventions très efficaces en s'appuyant sur l'emplacement des fichiers. Imaginez que l'on définisse pour notre passerelle de message une structure de dossiers présentée à la Figure 18.1.

---

1. Souvenez-vous que nous avons employé la même technique au Chapitre 13 pour simplifier une factory abstraite.

Figure 18.1
La structure des dossiers de la passerelle des messages



Cette structure de dossiers n'est pas particulièrement originale, elle est souvent utilisée dans des projets Ruby[1]. Une structure standard de répertoires peut nous aider à résoudre le problème du chargement d'adaptateurs, donc cette structure est particulièrement pertinente dans notre cas :

```
def load_adapters
  lib_dir = File.dirname(__FILE__)
  full_pattern = File.join(lib_dir, 'adapter', '*.rb')
  Dir.glob(full_pattern).each {|file| require file }
end
```

La méthode `load_adapters` déduit le chemin du dossier des adaptateurs en partant de la constante __FILE__. L'interpréteur Ruby affecte à cette constante le chemin du fichier source courant. Quelques manipulations des méthodes de la classe `File` nous permettent de trouver un patron de nom pour tous nos adaptateurs : `"adapter/* . rb"`. Ensuite, la méthode utilise ce patron pour rechercher et inclure toutes les classes d'adaptateurs. Cette approche fonctionne car `require` n'est qu'un appel de méthode Ruby comme un autre, et cet appel peut être effectué à partir du code à tout moment lorsque nous devons charger un fichier source dans l'interpréteur Ruby. Nous devons toutefois compléter notre convention : nommez votre classe `<protocol>Adapter` et placez-la dans le dossier `adapter`.

1. Un projet doit être organisé de cette façon pour être empaqueté en tant que gem, c'est probablement la raison pour laquelle cette structure est si répandue.

Tout ceci est assez remarquable car nous venons d'écrire en très peu de lignes tout ce qui est nécessaire au bon fonctionnement d'une passerelle de messages. Voici la classe `MessageGateway` complète :

```
class MessageGateway
  def initialize
    load_adapters
  end

  def process_message(message)
    adapter = adapter_for(message)
    adapter.send_message(message)
  end

  def adapter_for(message)
    protocol = message.to.scheme
    adapter_class = protocol.capitalize + 'Adapter'
    adapter_class = self.class.const_get(adapter_class)
    adapter_class.new
  end

  def load_adapters
    lib_dir = File.dirname(__FILE__)
    full_pattern = File. join(lib_dir, 'adapter', '*.rb')
    Dir.glob(full_pattern).each {|file| require file }
  end
end
```

Il suffit maintenant d'appeler la méthode `process_message` en lui passant un objet `Message`, et vous le verrez tout de suite partir joyeusement vers sa destination.

Il faut noter deux caractéristiques importantes dans la convention que nous venons d'établir. Premièrement, le seul but de cette convention est de faciliter l'ajout d'adaptateurs. L'objectif n'était absolument pas de rendre tous les aspects de la passerelle de messages facilement extensibles. Pourquoi ? Parce que nous avons supposé que nos utilisateurs, des ingénieurs logiciel, auraient besoin d'ajouter de nouveaux adaptateurs assez fréquemment. Si l'on anticipe ce besoin futur, on peut faciliter la vie des fournisseurs d'adaptateurs.

Deuxièmement, malgré le fait que notre convention impose quelques contraintes – l'auteur d'un adaptateur doit suivre une règle pour nommer son adaptateur et il doit le placer dans un dossier précis –, ces contraintes ne sont pas gênantes, car tout ingénieur consciencieux applique déjà ces règles.

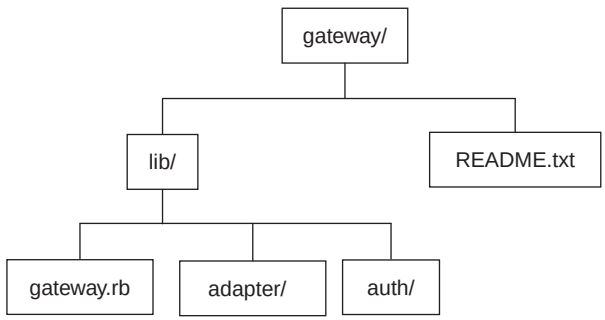
Ajouter un niveau de sécurité

Maintenant que la première version de la passerelle de messages est opérationnelle, il faut penser à ajouter un niveau de sécurité. Pour être précis, on voudrait appliquer une politique générale pour contrôler quels utilisateurs ont l'autorisation d'envoyer des messages à un hôte donné. Qui plus est, il faut pouvoir gérer un certain nombre d'utilisateurs particuliers, qui ne seront pas soumis à la politique générale pour un hôte donné.

On peut commencer par la mise en place d'une classe de contrôle d'accès pour chaque hôte de destination. Cette contrainte paraît acceptable si le nombre d'hôtes reste limité. La convention de nommage et l'emplacement des fichiers ont tellement bien fonctionné pour les adaptateurs que nous décidons d'adopter une approche similaire pour les classes de contrôle d'accès : nommez votre classe de contrôle d'accès `<destination_host>Authorizer` et placez-la dans le dossier `auth`.

La Figure 18.2 illustre notre structure de dossiers après la mise à jour.

Figure 18.2
La structure de dossiers de la passerelle étendue avec le contrôle d'accès



Le nommage des classes de contrôle d'accès soulève un problème car, en règle générale, les noms des hôtes ne correspondent pas à des noms de classes valides en Ruby. On devra recourir à la magie de la transformation de chaînes. Traduisons un nom comme `russolsen.com` en classe de contrôle d'accès `RussolsenDotComAuthorizer`[1] :

```
def camel_case(string)
  tokens = string.split('.')
  tokens.map! {|t| t.capitalize}
  tokens.join('Dot')
end
```

1. Pour rester simple, la méthode `authorizer_for` ne gère pas correctement les noms d'hôte qui incorporent des tirets. Évidemment, il suffit d'ajouter quelques expressions régulières bien choisies pour traiter tous les noms d'hôte possibles.

```
def authorizer_for(message)
  to_host = message.to.host || 'default'
  authorizer_class = camel_case(to_host) + "Authorizer"
  authorizer_class = self.class.const_get(authorizer_class)
  authorizer_class.new
end
```

Mais à quoi doit ressembler l'interface des classes de contrôle d'accès ? Souvenez-vous que pour chaque hôte donné il doit exister un certain nombre de règles applicables à presque tous les utilisateurs. Je dis "presque" car il peut y avoir plusieurs exceptions. On pourrait imaginer, par exemple, que tous les utilisateurs sont autorisés à envoyer des messages courts à l'adresse `russolsen.com`, mais que seul `"russ.olsen"` a le droit d'y envoyer des messages longs.

On pourrait adopter une convention qui stipule que si la classe de contrôle d'accès possède une méthode nommée `<user name>_authorized?`, cette méthode sera utilisée pour autoriser le message. On serait là aussi obligé de transformer le nom de l'utilisateur pour qu'il respecte les règles de nommage de méthodes. Dans le cas où cette méthode n'existerait pas, on appellerait alors la méthode générique `authorized?`. Voilà à quoi pourrait ressembler une classe de contrôle d'accès typique :

```
class RussolsenDotComAuthorizer
  def russ_dot_olsen_authorized?(message)
    true
  end

  def authorized?(message)
    message.body.size < 2048
  end
end
```

Le code qui met en œuvre cette convention est très simple. Tout d'abord, on obtient une instance de la classe de contrôle d'accès pour le message en cours de traitement. Ensuite, on récupère le nom de la méthode qui gère la politique spéciale concernant l'expéditeur du message. Enfin, on vérifie que l'objet de contrôle d'accès répond à cette méthode. Si c'est le cas, alors, on utilise cette méthode, sinon on appelle la méthode standard `authorized?` :

```
def worm_case(string)
  tokens = string.split('.')
  tokens.map! {|t| t.downcase}
  tokens.join('_dot_')
end

def authorized?(message)
  authorizer = authorizer_for(message)
  user_method = worm_case(message.from) + '_authorized?'
```

```
    if authorizer.respond_to?(user_method)
      return authorizer.send(user_method, message)
    end
    authorizer.authorized?(message)
  end
```

Et voici maintenant la version finale de notre convention concernant le contrôle d'accès : nommez votre classe de contrôle d'accès `<destination_host>Authorizer` et placez-la dans le dossier `auth`. Implémentez la politique générale pour l'hôte dans la méthode `authorize`. Si une politique spéciale existe pour un utilisateur donné, implémentez-la dans la méthode `<user>_authorized?`.

Aider un utilisateur dans ses premiers pas

Il existe un autre moyen d'aider un ingénieur qui étend notre passerelle : on peut lui mettre le pied à l'étrier à ses tout débuts. Nous avons noté plus tôt dans ce chapitre que l'un des principes de la conception efficace d'interfaces consiste à fournir à l'utilisateur des modèles et des exemples. D'une part, on pourrait leur proposer quelques exemples qui illustrent la création d'un adaptateur pour un protocole ou la création d'une classe de contrôle d'accès. D'autre part, on pourrait fournir un utilitaire pour générer un squelette de ce genre de classe.

Pourquoi ne pas offrir à nos utilisateurs un générateur d'adaptateurs ? Voici un script Ruby qui prépare les bases d'un adaptateur :

```
protocol_name = ARGV[0]
class_name = protocol_name.capitalize + 'Adapter'
file_name = File.join('adapter', protocol_name + '.rb')
scaffolding = %Q{
class #{class_name}
  def send_message(message)
    # Le code d'envoi d'un message
  end
end
File.open(file_name, 'w') do |f|
  f.write(scaffolding)
end
```

Si l'on place ce code dans un fichier nommé `adapter_scaffold.rb`, on peut l'appeler à l'aide de la ligne de commande suivante pour créer le squelette d'un adaptateur FTP :

```
ruby adapter_scaffold.rb ftp
```

Nous nous retrouvons avec une classe nommée `FtpAdapter` dans un fichier `ftp.rb` placé dans le dossier `adapter`.

On sous-estime souvent la valeur de ce type de générateurs. Pourtant, ces utilitaires sont précieux pour les nouveaux utilisateurs souvent surchargés d'information lorsqu'ils abordent un environnement inconnu et qui peinent à démarrer.

Récolter les bénéfices de la passerelle de messages

On pourrait continuer l'extension de notre passerelle de messages en ajoutant une étape de transformation du format des messages ou en créant un outil d'audit flexible qui puisse garder un suivi de certains messages. Mais arrêtons-nous et voyons ce que nous avons accompli. On a construit une passerelle de messages extensible sur deux points : l'ajout de nouveaux protocoles et la prise en compte de règles de contrôle d'accès y compris des règles spécifiques à certains utilisateurs. Aucun fichier de configuration n'est nécessaire pour notre système. Le programmeur qui souhaite étendre notre système doit simplement écrire la bonne classe et la placer dans le dossier approprié.

Un effet de bord intéressant et inattendu de l'utilisation de ces conventions réside dans la simplification du code principal de la passerelle. Si l'on avait opté pour l'usage de fichiers de configuration, il aurait fallu les rechercher, les lire et probablement corriger des erreurs avant de pouvoir paramétrer nos adaptateurs et nos classes de contrôle d'accès. Dans notre cas, rien ne retarde l'usage de nos adaptateurs et de nos classes de contrôle d'accès.

User et abuser du pattern Convention plutôt que configuration

Un des dangers des systèmes fondés sur des conventions consiste à définir des conventions incomplètes, ce qui limite la flexibilité de votre système. Par exemple, la transformation des noms d'hôte en noms de classes Ruby n'est pas effectuée avec énormément de soin dans notre passerelle de messages. Le code dans ce chapitre fonctionne correctement avec des noms d'hôte simples tels que `russolsen.com`, qui est correctement converti en `RussOlsenDotCom`. Mais, si l'on utilise dans notre système actuel un nom comme `icl-gis.com`, le programme va tenter de trouver la classe portant le nom illégal `Icl-gisDotComAuthorizer`. D'habitude, ce type de problème peut être résolu élégamment si l'on autorise nos classes à surcharger les conventions en cas de besoin. Dans notre exemple, on pourrait permettre aux classes de contrôle d'accès de surcharger la correspondance par défaut entre le nom d'hôte et la classe et spécifier les hôtes pour lesquels cette nouvelle règle s'applique.

Lorsqu'un système s'appuie fortement sur ce genre de conventions, son fonctionnement peut sembler magique aux nouveaux utilisateurs. C'est une autre source de soucis potentiels. Il est peut-être pénible d'écrire et de maintenir des fichiers de configuration, mais ils fournissent une sorte de plan de l'implémentation du système – un plan compliqué et difficile à interpréter, certes, mais un plan néanmoins. Inversement, un système fondé sur des conventions et bien conçu doit présenter ces détails opérationnels sous forme de documentation !

N'oubliez pas qu'au fur et à mesure de l'évolution de la magie des conventions vous aurez besoin de tests unitaires de plus en plus détaillés pour vous assurer que le comportement de vos conventions demeure bien... conventionnel. Il est extrêmement déroutant pour un utilisateur de travailler sur un système piloté par des conventions incohérentes ou défaillantes.

Convention plutôt que configuration dans le monde réel

Rails reste le meilleur exemple d'un système féru de conventions. En pratique, notre passerelle de messages s'est fortement inspirée des conventions mises en œuvre dans Rails. L'élégance de Rails tient en grande partie à son application cohérente des conventions. En voici quelques exemples :

- Si votre application Rails est déployée sur `http://russolsen.com`, alors, la requête `http://russolsen.com/employees/delete/1234` appelle par défaut la méthode `delete` de la classe `EmployeesController`. La valeur `1234` est passée dans la méthode en paramètre.
- Les résultats de cet appel au contrôleur sont traités par la vue définie dans le fichier `views/employees/delete. html.erb`.
- Les applications Rails utilisent typiquement ActiveRecord pour communiquer avec la base de données. Par défaut, la table nommée `proposals` (au pluriel) est gérée par la classe `Proposal` (singulier), qui réside dans un fichier nommé `proposal.rb` (en minuscules) dans le dossier `models`. Le champ `comment` de la table `proposals` devient comme par magie l'attribut `comment` d'un objet `Proposal`.
- Rails fournit tout un éventail de générateurs qui aident les utilisateurs à créer leurs premiers modèles, vues et contrôleurs.

Une application Rails classique est littéralement pétrie de conventions.

Mais Rails n'est pas le seul exemple de l'utilisation judicieuse des conventions dans le monde Ruby. RubyGems est un utilitaire de paquetage standard pour des applications Ruby. Il est relativement simple d'utilisation, surtout lorsque l'on suit ses conventions d'organisation des dossiers, comme nous l'avons fait dans l'exemple avec la passerelle de messages.

En conclusion

Dans ce chapitre, nous avons étudié le pattern Convention plutôt que configuration. Ce pattern stipule qu'on peut parfois rendre son système plus convivial si le code est conçu autour de conventions fondées sur le nommage de classes, de fichiers, de méthodes et l'organisation standardisée des dossiers. Cette technique rend vos programmes facilement extensibles : pour étendre le système il suffit d'ajouter un fichier, une classe ou une méthode correctement nommé.

Le pattern Convention plutôt que configuration tire parti des mêmes qualités de dynamique et de flexibilité de Ruby qui rendent possibles les deux autres patterns spécifiques à Ruby décrits dans ce livre. Tout comme le pattern Domain-Specific Language, Convention plutôt que configuration s'appuie principalement sur l'évaluation de code au moment de l'exécution. Tout comme le pattern Méta-programmation, pour bien fonctionner il requiert un niveau d'introspection relativement élevé de la part des programmes. Ces trois patterns partagent une autre caractéristique : leur façon d'aborder certains problèmes de programmation. Leur philosophie commune prescrit qu'il ne faut pas simplement utiliser un langage de programmation, mais plutôt le remanier en un outil plus adapté à la résolution de vos problèmes.

Conclusion

Nous avons parcouru un chemin considérable dans cet ouvrage – de nos débuts avec le pattern Template Méthode jusqu'au chargement dynamique de classes en respectant des conventions. En cours de route, nous avons découvert que la nature dynamique et le typage à la canard de Ruby modifient la façon d'approcher de nombreux problèmes de programmation. Lorsqu'il faut faire varier le comportement d'un algorithme profondément encapsulé dans une classe, on peut développer un objet Strategy, mais on a également la possibilité de passer simplement un bloc de code. L'implémentation de patterns comme Proxy et Decorator, qui s'appuient fortement sur le principe de la délégation, n'est plus un exercice pénible d'écriture de code comme avec d'autres langages. Les fonctions dynamiques et réflexives de Ruby nous permettent d'implémenter l'idée de fabrique de classes tout en sortant du cadre des limitations imposées par les patterns classiques Abstract Factory et Factory Method, fondés sur l'héritage. Les adaptateurs ne présentent plus de problème dans un langage qui permet d'ajuster l'interface d'un objet à la volée. Des itérateurs externes sont toujours possibles et on peut en trouver dans le code Ruby, mais les itérateurs internes les remplacent avantageusement et ils sont, de fait, omniprésents. La technique des langages spécifiques d'un domaine, dans sa variante DSL interne, permet de se servir de l'interpréteur Ruby en tant qu'analyseur syntaxique lorsqu'on écrit son propre interpréteur.

Tout ceci ne devrait pas vous surprendre. Bien que le livre *Design Patterns* du GoF représente un pas géant dans l'art d'écrire des programmes, plus de quinze années se sont écoulées depuis sa parution. Ce ne serait pas très flatteur pour notre profession si tant d'années plus tard on cherchait encore à résoudre exactement les mêmes problèmes avec exactement les mêmes techniques. De nombreux patterns originaux du GoF sont des solutions pérennes qui nous accompagneront encore très longtemps. Mais la programmation a une caractéristique commune avec la littérature : la traduction de *Roméo et Juliette* de l'anglais vers le français change les phrases, les formulations et l'atmosphère générale de l'œuvre. Juliette serait toujours jeune et belle, mais elle serait légèrement différente dans la version française. Un design pattern traduit vers un autre langage – Ruby – resterait le même pattern, mais il serait différent.

Lorsque l'on étudie les traductions en Ruby des motifs de conception répertoriés dans *Design Patterns*, on découvre que la plupart des différences résultent de la flexibilité quasiment illimitée de ce langage. En Ruby, lorsque le comportement ou l'interface d'une classe ne vous semble pas adaptée à la tâche, vous avez beaucoup d'options. Vous pouvez certainement encapsuler des instances de cette classe dans un adaptateur. Vous pouvez doter cette classe d'un décorateur ou d'un proxy ou bien créer une fabrique pour créer des instances encapsulées. Cette fabrique peut être implémentée comme un singleton. Si vous avez sous la main un objet complexe, probablement développé par une autre équipe dans votre grande organisation, tous ces choix peuvent être judicieux. Toutefois, lorsque vous gérez un objet simple que vous connaissez bien, vous avez la possibilité de le modifier directement et de lui apporter le comportement requis. Avec Ruby, nous ne sommes plus obligés de recourir à des design patterns très élaborés pour résoudre des problèmes locaux. Ruby nous fournit des outils pour faire simplement les choses simples.

S'il est une chose qui n'a pas changé depuis la publication de *Design Patterns*, c'est bien le besoin de l'expérience collective. Bruce Tate aime à rappeler[1] que, lorsqu'une nouvelle technique de programmation ou un langage apparaît, on observe souvent un retard dans l'adoption des bonnes pratiques d'utilisation. L'industrie a besoin de temps pour s'approprier la technique et élaborer les meilleurs moyens de l'exploiter. Pensez aux années qui se sont écoulées entre la prise de conscience que la programmation orientée objet était la voie à suivre et le moment où l'on a effectivement commencé à appliquer cette technologie efficacement. Cet intervalle, c'est le fameux "temps d'expérience" nécessaire pour accumuler les bonnes techniques orientées objet.

La reconnaissance de plus en plus large des avantages offerts par les langages dynamiques tels que Ruby nous plonge dans cette nouvelle phase d'acquisition d'expérience. Les fonctions puissantes de Ruby suggèrent de nouvelles approches des problèmes de programmation avec lesquels on lutte depuis des années. Ruby nous fournit également les moyens de réaliser des choses qui étaient impossibles ou très difficiles jusqu'alors. Mais que devons-nous faire ? Quels raccourcis peut-on prendre sans danger ? Quels pièges faut-il éviter ? Ruby met à notre disposition toute cette puissance, mais nous avons besoin de conseils – et d'expérience – pour l'accompagner. Dans ce livre, j'ai tenté d'apporter quelques éclaircissements sur la façon de canaliser la puissance de Ruby. Au fur et à mesure que nous traversons cette phase d'expérimentation, de nouvelles solutions et de nouveaux patterns vont surgir qui s'inscriront mieux dans le monde dynamique et flexible de Ruby. Je ne sais pas à quoi ressembleront ces patterns, mais j'attends leur apparition avec impatience. Je sais aussi que c'est une période formidable pour être un développeur.

---

1. Voici un exemple de ces propos : http://weblogs.java.net/blog/batate/archive/2004/10/time_wisdom_and.html.

Annexes

A

Installer Ruby

L'une des caractéristiques d'un langage populaire est qu'il n'est pas difficile à trouver. Le bon endroit pour commencer vos recherches se trouve sur la page d'accueil du site Web dédié au langage Ruby, située sur **http://www.ruby-lang.org**. La suite dépend du système d'exploitation que vous utilisez.

Installer Ruby sous Microsoft Windows

Si vous utilisez Microsoft Windows, optez pour l'installeur "en-un-clic" de Ruby, qui se trouve à cette adresse : **http://rubyforge.org/projects/rubyinstaller**. Ce programme installe sur votre machine l'environnement de base Ruby ainsi que tout un tas d'utilitaires pratiques. La procédure ne demande que quelques clics de souris. N'oubliez pas d'activer l'option RubyGems pour installer le gestionnaire de paquetages logiciels de Ruby.

Si vous êtes plus orienté UNIX, mais que vous utilisiez néanmoins un poste de travail Windows, jetez un œil sur Cygwin (**http://www.cygwin.com**), un environnement à la UNIX pour Windows, qui inclut Ruby dans sa distribution.

Installer Ruby sous Linux ou un autre système de type UNIX

Si vous utilisez un système de type UNIX tel que Linux, vous avez plusieurs options :

- Installer une distribution toute faite. Il est très probable qu'une distribution de Ruby existe pour votre système. N'oubliez pas d'installer également RubyGems pour récupérer le gestionnaire de paquets Ruby. Si votre système est sous Debian Linux ou un de ses dérivés (ce qui inclut Ubuntu Linux, aujourd'hui très populaire), sachez que RubyGems n'est pas disponible comme un paquetage Debian préconstruit à

cause de différences philosophiques sur la façon d'empaqueter les logiciels. Dans ce cas, passez à la construction de RubyGems à partir du code source.

- Construire Ruby à partir du code source. Construire votre environnement Ruby à partir du code source n'est pas très difficile. Il suffit de télécharger le logiciel et de suivre les instructions présentes dans le fichier README. Lorsque l'installation de Ruby est terminée, récupérez et construisez de la même façon le code source de RubyGems.

Mac OS X

Bonne nouvelle : OS X Tiger est livré avec Ruby déjà préinstallé. La mauvaise nouvelle, c'est qu'il s'agit d'une vieille version de Ruby. Beaucoup d'utilisateurs de Ruby sous OS X – probablement la majorité – préfèrent construire Ruby à partir du code source (voir la section précédente) ou bien récupérer Ruby sur le site MacPorts (**http://www.macports.org/**).

La version Leopard de Mac OS X qui vient de sortir a considérablement amélioré le support de Rubyet il y a plus de bonnes nouvelles que de mauvaises.

B

Aller plus loin

Une littérature énorme portant sur les design patterns est apparue au cours des quinze dernières années, et la littérature sur Ruby augmente de jour en jour. Cette annexe indique certaines ressources qui peuvent être utiles au programmeur qui s'intéresse à la fois à Ruby et aux design patterns.

Design patterns

Évidemment :

> Gamma E., Helm R., Johnson R. et Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley, 1995.

J'apprécie toujours les œuvres originales, et si vous êtes intéressé par les design patterns il n'y a rien de mieux que le *Design Patterns* du *Gang of Four.*

Un choix, moins évident, mais qui vaut le détour :

> Alpert S., Brown K. et Woolf B., *The Design Patterns Smalltalk Companion*, Reading, MA: Addison-Wesley, 1998.

Mes collègues qui travaillent avec Smalltalk me rappellent sans cesse que ce langage de programmation possède tous les points positifs que nous découvrons aujourd'hui dans Ruby. Et ces points existent depuis des décennies. Le fait que Smalltalk n'ait pas gagné une large popularité est probablement dû à sa syntaxe étrange plutôt qu'à un manque de puissance ou d'élégance. Mais ne parlons plus des différences des langages, *The Design Patterns Smalltalk Companion* vaut la peine d'être lu car c'est une étude soigneuse de l'application des design patterns dans un langage tout aussi dynamique et flexible que Ruby.

Pour un regard plus récent sur les mêmes thèmes on se portera sur :

> Sweat J., *php|architect's Guide to PHP Design Patterns*, Toronto, ON: Marco Tabini and Associates, 2005.

Il existe énormément de littérature concernant les design patterns dans des langages différents, notamment en Java. Voici deux livres intéressants qui se concentrent sur Java :

> Freeman E., Freeman E., Bates et Sierra K., *Head First Design Patterns*, Sebastopol, CA: O'Reilly Media, 2004.

> Stelting S. et Maassen O., *Applied Java Patterns*, Palo Alto, CA: Sun Microsystems Press, 2002.

Ces deux livres sont très différents : *Applied Java Patterns* est un ouvrage très détaillé et plus traditionnel, alors que *Head First Design Patterns* contient moins de détails mais il est bien plus ludique.

Ruby

Le meilleur ouvrage d'introduction à Ruby est celui écrit par Dave Thomas, Chad Fowler et Andy Hunt :

> Thomas D., Fowler C. et Hunt A., *Programming Ruby: The Pragmatic Programmers' Guide*, seconde édition, Raleigh, NC: The Pragmatic Bookshelf, 2005.

Programming Ruby est une présentation complète de Ruby, son environnement et ses bibliothèques. Néanmoins, je dois admettre que lorsque je cherche une analyse vraiment profonde des questions liées à Ruby j'ai tendance à ouvrir le livre suivant :

> Black D., *Ruby for Rails*, Greenwich, CT: Manning Publications, 2006.

Soyez conscient que, malgré son nom, *Ruby for Rails* se concentre à 85 % sur Ruby et seulement à 15 % sur Rails.

Ruby est en grande partie un langage idiomatique dans lequel il existe une véritable façon de s'exprimer en langage Ruby. Dans ce livre, j'ai essayé d'indiquer la façon d'aborder des problèmes selon la philosophie Ruby. Si vous souhaitez mieux comprendre cette philosophie, lisez :

> Fulton H., *The Ruby Way*, seconde édition, Boston: Addison-Wesley, 2006.

The Ruby Way est en partie un répertoire de solutions techniques et en partie un texte d'introduction. Si vous voulez savoir quelles techniques appliquer pour accomplir certaines tâches en Ruby et si vous souhaitez avoir un peu d'information sur le pourquoi de ces techniques, ce livre est fait pour vous.

Un ouvrage du même genre :

> Carlson L. et Richardson L., *Ruby Cookbook*, Sebastopol, CA: O'Reilly Media, 2006.

Malgré mon penchant pour les livres, je reconnais que lorsqu'on apprend un nouveau langage de programmation il existe une ressource tout aussi importante que la littérature : des bons programmes écrits dans ce langage. Si vous souhaitez sérieusement apprendre Ruby, vous devez passer du temps à étudier les codes source suivants :

- La bibliothèque standard de Ruby. C'est la totalité du code livré avec votre distribution de Ruby. Vous êtes curieux à propos de la classe `Complex` ? Vous avez des questions sur `Webrick` ? Vous voulez tout savoir sur `URI` ? Il suffit de jeter un œil, car tout ce code est sur votre disque dur.
- Ruby on Rails. N'hésitez pas à étudier le code source de l'application vedette. Toutefois, la plus grande partie de Rails est écrite en Ruby très avancé. Ne soyez pas intimidé mais attendez-vous à vous poser fréquemment la question : "Comment est-ce que ce truc fonctionne ?" Le site Web de Rails se trouve à **http://www.rubyonrails.org**.
- Ruby Facets est une collection énorme d'utilitaires Ruby. Ces utilitaires sont en réalité des extensions des classes standard de Ruby, ce qui présente un intérêt particulier pour les débutants en Ruby. Cette ressource est bien intéressante et très utile. Le site Web de Facets se trouve à l'adresse **http://facets.rubyforge.org**.

Expressions régulières

Les expressions régulières ont été mentionnées plusieurs fois dans ce livre. Si vous n'avez pas encore trouvé le temps d'apprendre cet outil terriblement pratique, je vous recommande de le faire en priorité. Commencez par :

> Friedl J., *Mastering Regular Expressions*, Sebastopol, CA: O'Reilly Media, 2006.

Blogs et sites Web

Le site Web principal de Ruby est **http://www.ruby-lang.org**. La plupart des gens qui s'intéressent à Ruby seront aussi curieux de Rails, qui se trouve à l'adresse **http://www.rubyonrails.org**. Le portail francophone de référence, Railsfrance, se trouve quant à lui à l'adresse **http://www.railsfrance.org**.

Il existe un certain nombre de bons blogs sur Ruby et Rails. En voici quelques-uns que je trouve particulièrement utiles et intéressants :

- Jamis Buck's blog – **http://weblog.jamisbuck.org**.
- Jay Fields's blog – **http://blog.jayfields.com**.
- Ruby Inside – **http://www.rubyinside.com** (où Peter Cooper nous livre un condensé de sa grande sagesse).

Le site Web associé à ce livre est **http://designpatternsinruby.com**. Enfin, si vous voulez avoir de mes nouvelles, visitez **http://www.russolsen.com** ou envoyez-moi un message à russ@russolsen.com.

À propos des traducteurs

Laurent Julliard est actuellement directeur associé de la société Nuxos, SSLL spécialisée dans le conseil, le développement et la formation Ruby & Rails. Précédemment chef de projet et architecte logiciel dans les divisions R&D de grands groupes comme HP et Xerox, il est aussi pionnier de l'Open Source en France, fondateur du premier groupe d'utilisateur Linux en France (GUILDE) dès 1995 et membre actif de la communauté Ruby depuis début 2000.

Laurent Julliard a participé à plusieurs projets Ruby d'importance (dont l'environnment de développment intégré FreeRIDE). En coopération avec Richard Piacentini, il est aussi le traducteur d'ouvrages de référence et l'auteur de plusieurs articles sur Ruby et Rails. Il intervient régulièrement en tant que conférencier sur Ruby et Rails et notamment lors de la conférence annuelle "Paris on Rails" dont il est co-organisateur.

Mikhail Kachakhidze est traducteur technique multilangues depuis 1996. Il compte à son actif de nombreux projets de localisation de logiciels, sites web et documentation. Il a contribué entre autres aux versions russes d'Oracle 8i, Windows XP, MS SQL Server 2005.

En 2004, Mikhail Kachakhidze passe du côté obscur de la force et devient développeur. Après quelques expériences en Java, il adopte Ruby et Rails. Aujourd'hui, il fait partie de l'équipe de développement de la société Eyeka.

Richard Piacentini côtoie les technologies du libre depuis le début de sa carrière dans des domaines allant de la gestion d'information à flux tendu (TF1/LCI, Tempost-La Poste) à la modélisation comportementale (eGoPrism) en passant par les communautés de pratiques (Alphanim, Libération) ou la gestion de systèmes de production infographique en réseaux (INA, France Animation).

Initiateur et organisateur, avec Laurent Julliard, de la conférence "Paris on Rails", il est le fondateur de Nuxos Group, SSLL spécialiste de Ruby et Rails, ainsi que le créateur du portail Railsfrance. Il a traduit plusieurs ouvrages de référence et dispense régulièrement des formations en Europe et en Afrique du Nord. Richard Piacentini est un fervent adepte des bonnes pratiques du développement logiciel et il traumatise régulièrement les développeurs et stagiaires qui l'entourent en les persuadant d'écrire du code efficace et élégant ainsi que d'innombrables tests unitaires et fonctionnels.

Index

Symboles

!, opérateur 28
!~, opérateur 38
&, opérateur 28
&&, opérateur 28
+, opérateur 32
<, opérateur 27
<<, opérateur 36
<=, opérateur 27
<=>, opérateur 82
==, opérateur 27
=~, opérateur 37
>, opérateur 27
>=, opérateur 27
|, opérateur 28
||, opérateur 28

A

abs, méthode 26, 144
Abstract Factory 16
Abstract Factory, pattern 196, 205
  lots d'objets 205
  nommage 208
Accesseur 40
  attr_accessor 41
  attr_reader 42
  pattern Méta-programmation 266
ActiveRecord
  méthodes magiques 222
  migration 134
  pattern Adapter 148
  pattern Command 134
  pattern DSL 255
  pattern Factory Method 210
  pattern Méta-programmation 268
  pattern Observer 94
  relation 268
ActiveSupport
  pattern Decorator 177
  pattern Singleton 193
Adapter, pattern 15, 139
  ActiveRecord 148
  adapter ou modifier 147
  alternatives 144
  fichier 276
  FTP 283
  HTTP 276
  modifier une instance unique 145
  sélectionner 277
  SMTP 276
Addition 24
alias, mot clé 175
alias_method_chain, méthode 177
all?, méthode 116
and, opérateur 28
any?, méthode 116
Arbre de syntaxe abstrait (AST) pour analyseurs syntaxiques
  complexe 238
  développer 226
  interpréteur de recherche de fichiers 229
  interpréteur sans analyseur 236
  simple 234
  XML et YAML 237

Argument 43
astérisque 44
nombre arbitraire 44
valeurs par défaut 44
Array, classe 35
AST *Voir* Arbre de syntaxe abstrait
attr_accessor 41
attr_reader 42
attr_writer 42

B

Base, classe 210
begin, instruction 47
Bignum, classe 24
Bloc *Voir* Proc, classe
Boucle 30
break 31
each 31
for 31
next 31
until 30
while 30
break, opérateur 31
Builder, pattern 16, 215
director 216
MailFactory 222
méthodes magiques 220
polymorphisme 216
produit 216
validation 219

C

call, méthode 75
Chaîne de caractères 32
+ 32
à plusieurs lignes 33
concaténation 32
downcase 32
each_byte, scan 119
immuable 34
interpolation 33
length 32
modifiable 34
String 119
upcase 32
class, méthode 26
Classe 38
définition 38
Command, pattern 15, 125
ActiveRecord 134
annulation, rollback 130
bloc 126
files de commandes 132
FXRuby 134
Madeleine 135
Comparaison 28
< 27
<= 27
<=> 82
== 27
> 27
>= 27
Composite, pattern 15, 97, 100
AST 227
composant 100
composite 99, 101, 105
création 100
feuille 100, 105
FXRuby 108
opérateurs 104
pointeurs 106
tableau 104
Composition 7
Console interactive irb 20
Constante 24
Constructeur 39
Convention plutôt que configuration, pattern 17, 271
chargement de classes 278
exemple 275
historique 272
pattern Adapter 277
principes 273
Rails 285
RubyGems 286

D

Decorator, pattern 16, 167
- ActiveSupport 177
- délégation 173
- forwardable 174
- héritage 169
- méthodes d'encapsulation 174
- module 175

def, mot clé 39
Délégation 11, 69
Design pattern 14, 293
Distributed Ruby (drb), Proxy 164
Division 24
Domain-Specific Language (DSL) *Voir* **DSL, pattern**
downcase, opérateur 32
DSL, pattern 17, 245
- ActiveRecord 255
- analyseur syntaxique 246
- avantages 251
- externe 246
- interne 246
- interpréteur 246
- rake 254

Duck typing 61

E

each, méthode 119
each, opérateur 31
each_byte, méthode 119, 120
each_entry, méthode 121
each_filename, méthode 120
each_key, méthode 119
each_line, méthode 120
each_object, méthode 121
each_value, méthode 119
else, opérateur 29
elsif, opérateur 29
end, opérateur 29, 30
Entier 24
Enumerable, module 116
Etc, module 154
Exception 47
- begin, rescue 47
- NoMethodError 159
- raise 48

Expression régulière 37, 295
extend, méthode. 175

F

Factory Method, pattern 16, 196, 199
- ActiveRecord 210
- paramètres 201

false 28
FalseClass, classe 28
Fichier source 49
- chargement 49
- require 49

File.basename, méthode 231
Fixnum, classe 24, 144
float 25
for, opérateur 31
Forwardable, module 267
forwardable, module 174
FXRuby 108, 134
- pattern Command 134
- pattern Composite 108

G

Gang of Four 4
GenericServer, classe 67

H

Hash, classe 119
Hello World 20
Héritage 7, 42
- pattern Decorator 169
- super 43
- Template Methode 69

I

if, opérateur 29
include, instruction 45
Inflections, classe 194
initialize, méthode 39
Installer Ruby
  Linux, Unix 291
  Mac OS X 292
  Microsoft Windows 291
Interface 6
Interpolation 33
Interpreter, pattern 16, 226
  analyseur syntaxique 226
  AST 226
  contexte 228
  noeuds non terminaux 227
  noeuds terminaux 227
  Runt 240
IO, classe 120, 147
  each_byte 120
  each_line 120
  pattern Itérateur 120
irb 20
Itérateur, pattern 15, 111
  Enumerable 116
  externe 112
  interne 113
  IO 120
  Java 112
  ObjectSpace 121
  tableau 119

J

join, méthode 49

L

lambda, méthode 75
Langage spécifique d'un domaine *Voir* DSL, pattern
length, méthode 32

M

Madeleine, pattern Command 135
MailFactory 222
Méta-design patterns 5
Méta-programmation, pattern 17, 257
  ActiveRecord 268
  attr_accessor 266
  attr_reader 266
  attr_writer 266
  Forwardable 267
  pattern Composite 261
  pattern DSL 269
  public_method 265
  réflexion 264
  respond_to? 265
method_missing, méthode 159, 162
Mixin 47
Module 45
  définition 45
  Enumerable 116
  Etc 154
  extend 175
  Forwardable 267
  forwardable 174
  include 45
  mixin 47
  Module 267
  ObjectSpace 121
  pattern Méta-programmation 259
  Singleton 184, 189
Module, module 267
Multiplication 24
Mutateur 40
  attr_accessor 41
  attr_writer 42
  pattern Méta-programmation 266

N

new, méthode 39
next, opérateur 31

nil 27
nil?, méthode 26
NilClass, classe 27
Nombre à virgule flottante 25
not, opérateur 28

O

Object, classe 26
ObjectSpace, each_object 121
ObjectSpace, module 121
Objet sur mesure
méthode 258
module 259
pattern Méta-programmation 258
Observable, module 90
Observer, pattern 15, 87
bloc 91
couplage 86
Java 88
Observable 90
observateur 87
pull 92
push 92
sujet 87
Opérateurs booléens 27
Opérations arithmétiques 24
or, opérateur 28

P

Pathname, classe 120
each_entry 121
each_filename 120
pattern 16
Philosophie Ruby 22
Prevayler 135
private, mot clé 188
Proc, classe 75, 122
accolades 76
call 75
convention 76
création 75
do/end 76
lambda 75
paramètres 76
pattern Commande 126
pattern Itérateur 122
yield 77
Proxy, pattern 15, 151
contrôle d'accès 153
Distributed Ruby 164
method_missing 159
proxy distant 155
proxy virtuel 156
sujet 152
transfert des messages 158
public_methods, méthode 265

R

Racc 238
Rails
ObjectSpace 122
pattern Convention plutôt que configuration 271, 285
raise, instruction 48
rake
pattern DSL 254
pattern Singleton 194
rdoc
RIGenerator 81
Strategy 80
require, instruction 49
rescue, instruction 47
respond_to?, méthode 265
reverse, méthode 36
reverse!, méthode 36
reverse_each, méthode 119
REXML,Observer 94
round, méthode 26
RPC 155

Ruby 16, 294
RubyGems 50, 291
  gem 279
  pattern Convention plutôt que configuration 286
run, méthode 67
Runt, Interpreter 240

S

scan, méthode 119
Sécurité de type 61
self, variable 42, 181
send, méthode 160
Singleton, module 184
Singleton, pattern 179
  ActiveSupport 193
  classes 187
  instanciation immédiate 185
  instanciation tardive 185
  Java 179
  module 184
  modules en tant que singletons 188
  test 193
  variables globales 186
size, opérateur 35
SOAP 155, 209
sort, méthode 36
sort!, méthode 36
Soustraction 24
Strategy, pattern 15, 71
  composition, délégation 72
  contexte 71
  rdoc 80
  vs. Template Method 82
String, classe *Voir* Chaînes de caractères
succ, méthode 26
super, méthode 43
super(), méthode 90
Symbole 34

T

Tableau 35
  Array 35
  each 119
  length 35
  pattern Composite 104
  pattern Itérateur 119
  reverse 36
  reverse! 36
  reverse_each 119
  size 35
  sort 36
  sort! 36
  trie 81
Tableau associatif 36
  création 37
  each_key 119
  each_value 119
  Hash 119
  initialisation 37
  symbole 37
Template Method, pattern 15, 58, 199
  défauts, héritage 69
  hook 60
  méthodes d'accrochage 59
  WEBrick 67
Temps d'expérience 288
Test 64
  JUnit, NUnit, XUnit 64
  pattern Singleton 193
  setup, teardown, assert_equal, assert_not_nil 65
Thread
  création, exécution 48
  join 49
Thread, classe 48
Tk 134
to_s, méthode 26, 163
true 28
TrueClass, classe 28
truncate, méthode 26
Typage à la canard 61

Typage dynamique 61, 147
Strategy 74
vs. typage statique 63
Type primitif 26

U

unless, opérateur 30
until, opérateur 30
upcase, opérateur 32

V

Variable 22
$ 185
@ 39
accesseur 40
affectation 23
constante 24
d'instance 39
de classe 180
déclaration 23
globale 185
mutateur 40
nommage 23
ordinaire 22
self 181
Vous n'aurez pas besoin de ça 12, 209

W

WEBrick
GenericServer 67
Template Method 67
while, opérateur 30

X

XML 237
XMLRPC 209
XOR 140

Y

YAGNI ***Voir*** **Vous n'aurez pas besoin de ça**
YAML 237
yield, mot clé 77

Référence

Les design patterns en Ruby

Abordez les design patterns sous l'angle Ruby !

La plupart des livres consacrés aux design patterns sont basés sur C++ et Java. Mais le langage Ruby est différent et les qualités uniques de ce langage rendent l'implémentation et l'utilisation des patterns plus simples. Russ Olsen démontre dans ce livre comment combiner la puissance et l'élégance des design patterns pour produire des logiciels plus sophistiqués et efficaces avec beaucoup moins de lignes de code.

Il passe en revue du point de vue Ruby quatorze des vingt-trois patterns classiques du livre de référence produit par le fameux « Gang of Four » (problèmes résolus par ces patterns, analyse des implémentations traditionnelles, compatibilité avec l'environnement Ruby et améliorations spécifiques apportées par ce langage). Et vous apprendrez comment implémenter des patterns en une ou deux lignes de code là où d'interminables lignes de code sans intérêt sont nécessaires avec d'autres langages plus conventionnels.

Vous y découvrirez également de nouveaux patterns élaborés par la communauté Ruby, en particulier la métaprogrammation qui permet de créer des objets sur mesure ou le très ambitieux pattern « Convention plutôt que configuration » popularisé par Rails, le célèbre framework de développement d'applications web écrit en Ruby.

Passionnant, pratique et accessible, le livre *Les design patterns en Ruby* vous aidera à développer des logiciels de meilleure qualité tout en rendant votre expérience de la programmation en Ruby bien plus gratifiante.

TABLE DES MATIÈRES

- Améliorer vos programmes avec les patterns
- Démarrer avec Ruby
- Varier un algorithme avec le pattern Template Method
- Remplacer un algorithme avec le pattern Strategy
- Rester informé avec le pattern Observer
- Assembler le tout à partir des composants avec Composite
- Accéder à une collection avec l'Itérateur
- Effectuer des actions avec Command
- Combler le fossé avec l'Adapter
- Créer un intermédiaire pour votre objet avec Proxy
- Améliorer vos objets avec Decorator
- Créer un objet unique avec Singleton
- Choisir la bonne classe avec Factory
- Simplifier la création d'objets avec Builder
- Assembler votre système avec Interpreter
- Ouvrir votre système avec des langages spécifiques d'un domaine
- Créer des objets personnalisés par méta-programmation
- Convention plutôt que configuration

À propos de l'auteur

Russ Olsen est développeur de logiciels depuis plus de vingt-cinq ans. Il a géré des projets de développement à travers plusieurs générations de technologies de programmation : de FORTRAN à Ruby en passant par C, C++ et Java. Il utilise et enseigne Ruby depuis 2002 et est l'auteur d'un blog technique très lu, Technology As If People Mattered (www.russolsen.com).

Programmation

Niveau : Intermédiaire / Avancé
Configuration : Multiplate-forme

PEARSON

Pearson Education France
47 bis, rue des Vinaigriers
75010 Paris
Tél. : 01 72 74 90 00
Fax : 01 42 05 22 17
www.pearson.fr

ISBN : 978-2-7440-4018-4