6

Le refactoring avec les design patterns

Les design patterns, ou modèles de conception réutilisables, apportent des solutions génériques à des problèmes récurrents rencontrés dans le développement de logiciels. Ces modèles sont indépendants des langages mais reposent pour la plupart sur l'approche orientée objet.

Un certain nombre de design patterns ont été popularisés en 1995 par l'ouvrage collectif *Design patterns : catalogue de modèles de conception réutilisables,* de Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, plus connus sous le nom de Gang of Four, ou GoF. Si, depuis lors, de nombreux autres design patterns ont été formalisés, un grand nombre d'entre eux dérivent des travaux du GoF.

De par leur nature, les design patterns visent à introduire au sein des logiciels les meilleures pratiques de conception et peuvent être employés dans les projets de refactoring pour améliorer la qualité de l'existant.

Nous commençons par donner une description synthétique de ces modèles de conception réutilisables puis présentons l'utilisation de quelques design patterns du GoF particulièrement utiles dans le cadre d'un projet de refactoring.

Les design patterns

Les design patterns, en français modèles de conception, constituent une des matérialisations de ce concept fort de la programmation orientée objet qu'est la réutilisation.

Pour reprendre une image évoquée par Jim Coplien dans *Software Patterns,* les design patterns peuvent être vus un peu comme des patrons de couturière. La finalité d'un patron est de faire un habit, qui est le problème à résoudre. Les caractéristiques de cet habit constituent le contexte, et les instructions du patron la solution.

Pour être facilement utilisable, un design pattern doit être correctement documenté. Dans les différents catalogues de design patterns, celui du GoF étant le plus connu, chacun d'eux est généralement décrit au travers de plusieurs rubriques, notamment les suivantes :

- nom du design pattern ;
- description du problème concerné par le design pattern ;
- description de la solution (structure, collaborations) et de ses éventuelles variantes ;
- résultat obtenu avec ce design pattern et éventuelles limitations ;
- exemples ;
- design patterns apparentés.

Grâce à cette documentation, la mise en œuvre des design patterns est à la portée de tous et permet, à moindre coût, d'intégrer les meilleures pratiques au sein des développements. Il reste bien entendu nécessaire de comprendre le fonctionnement et le domaine d'application d'un design pattern pour bien l'utiliser.

Afin d'illustrer notre propos, prenons l'exemple d'un des design patterns les plus simples du GoF, le singleton :

- **Le problème.** Certaines classes ne doivent pas avoir plus d'une instance lors de l'exécution du programme auquel elles appartiennent. Cela se justifie soit par la nature de la classe (elle modélise un objet unique, comme un ensemble de variables globales à l'application), soit par souci d'économie de ressource mémoire (une instance unique fournit le même niveau de service que de multiples instances).
- **La solution du problème.** La classe doit comporter un attribut statique, généralement appelé `instance`, destiné à recevoir la référence de l'instance unique pour l'ensemble du logiciel, et une méthode, généralement appelée `getInstance`, renvoyant la valeur d'`instance`. Si `instance` est vide, `getInstance` crée une nouvelle instance de la classe en la stockant dans l'attribut `instance` et la renvoie à l'appelant.

Le code ci-dessous montre un exemple d'implémentation d'un singleton en Java :

```
public class MySingleton {
    //...
    private static MySingleton instance = null;

    protected MySingleton() {
        //...
    }

    public static MySingleton getInstance() {
        if (instance==null) {
            instance = new MySingleton();
        }
        return instance;
    }
    //...
}
```

Grâce à la méthode `getInstance`, les classes utilisatrices de la classe `MySingleton` sont certaines d'utiliser la même instance de cette dernière. L'inconvénient est que l'opérateur `new` n'est pas utilisable par les appelants. L'application du design pattern singleton à la classe `MySingleton` n'est donc pas transparente pour les classes utilisatrices, puisqu'elles doivent appeler en lieu et place du constructeur la méthode `getInstance`.

Afin d'éviter la création d'instances de `MySingleton` par d'autres biais que la méthode `getInstance`, nous déclarons son constructeur avec une portée protégée. Ainsi, seuls les descendants de `MySingleton` sont en mesure d'utiliser directement ce constructeur.

Mise en œuvre des design patterns dans le cadre du refactoring

La formalisation des design patterns utilisée habituellement est conçue de manière à faciliter la phase de conception d'un logiciel. Elle ne prend pas en compte les contraintes liées à leur application sur du code existant.

Il est donc nécessaire de compléter cette formalisation pour pouvoir étendre la portée des design patterns au refactoring, notamment pour les critères suivants :

- **Gains attendus et risques à gérer.** Il est nécessaire de contrebalancer les gains par les risques encourus. En fonction du contexte, il préférable de ne pas appliquer un design pattern si les gains attendus sont faibles en comparaison des risques pris.
- **Moyens de détection des cas d'application.** La partie dédiée à la description du problème adressé par le design pattern et à son contexte est souvent insuffisante pour détecter les cas d'application dans du code existant.
- **Modalités d'application et tests associés.** Dans le cadre d'un projet de refactoring, les modalités d'application d'un design pattern diffèrent de leur formalisation classique puisqu'ils modifient du code existant. Les tests de non-régression doivent donc être intégrés à la démarche afin de garantir le succès de la refonte.

Les design patterns du GoF sont regroupés selon les trois modèles suivants :

- **Modèles créateurs.** Design patterns spécialisés dans la création d'objets. Les plus connus sont la fabrique et le singleton.
- **Modèles structuraux.** Design patterns définissant différentes structures permettant la composition d'objets au-delà de la technique d'héritage. Les plus connus sont l'adaptateur et la façade.
- **Modèles comportementaux.** Design patterns proposant des structures de classes remarquables pour modéliser des comportements au sein des logiciels. Les plus connus sont la chaîne de responsabilité et l'observateur.

Dans les sections suivantes, nous nous concentrons sur les modèles comportementaux et structuraux du GoF, dont les bénéfices dans le cadre d'un refactoring sont les plus évidents.

Les modèles créateurs sont volontairement écartés, car, hormis pour le singleton, les cas d'utilisation ne sont pas aussi clairs que dans les autres modèles. Le lecteur intéressé pourra s'inspirer de notre démarche pour introduire ces design patterns dans ses projets de refonte.

Utilisation des modèles comportementaux

Les modèles comportementaux sont particulièrement intéressants du fait que leur application est généralement bien circonscrite, ce qui diminue les effets de bord et facilite les tests de non-régression.

Le pattern observateur

Au cours de sa vie, un objet peut être amené à changer plusieurs fois d'état. Au niveau de l'application, ces changements d'état génèrent des traitements. Par exemple, pour un traitement de texte, la modification du fichier ouvert doit activer la fonctionnalité d'enregistrement.

Si la classe de notre objet a la charge d'effectuer tous les traitements liés à ces changements d'état, nous pouvons arriver rapidement à une classe obèse, difficile à maintenir.

Le design pattern observateur permet à un objet de signaler un changement de son état à d'autres objets, appelés observateurs. Ce design pattern est particulièrement adapté à la programmation d'IHM graphiques. Tout contrôle graphique signale ses différents changements d'état au travers d'événements capturés par divers objets pour réaliser leurs traitements propres. Par exemple, dans un logiciel de traitement de texte, le clic sur un bouton de la barre d'outils déclenche des traitements qui sont pris en charge non pas directement par celui-ci, mais par ses observateurs.

Le design pattern observateur est simple à implémenter avec Java. L'API standard de J2SE fournit une interface (`java.util.Observer`) et une classe (`java.util.Observable`) offrant la base nécessaire.

L'interface `Observer` doit être implémentée par les classes des observateurs. Cette interface ne comprend qu'une seule méthode, `update`, qui est appelée pour notifier l'observateur d'un changement au niveau du sujet d'observation.

La classe `Observable` doit être utilisée par la classe observée. Cette classe fournit la mécanique d'inscription et de désinscription des observateurs (méthodes `addObserver` et `deleteObserver`) ainsi que la mécanique de notification (méthodes `notifyObservers`).

L'utilisation d'`Observable` par la classe observée peut se faire de deux manières. La première consiste à employer l'héritage, avec toutes les contraintes que cela impose (impossibilité d'hériter d'autres classes). La seconde consiste à créer une classe interne héritant d'`Observable`. Cette dernière manière nous semble plus adaptée à un contexte de refactoring. Elle offre de surcroît davantage de flexibilité dans le cas où la classe observée comporte plusieurs sujets d'observation (il suffit de créer une classe interne par sujet).

Le schéma UML de la figure 6.1 illustre l'utilisation de ce design pattern dans le cadre d'une refonte.

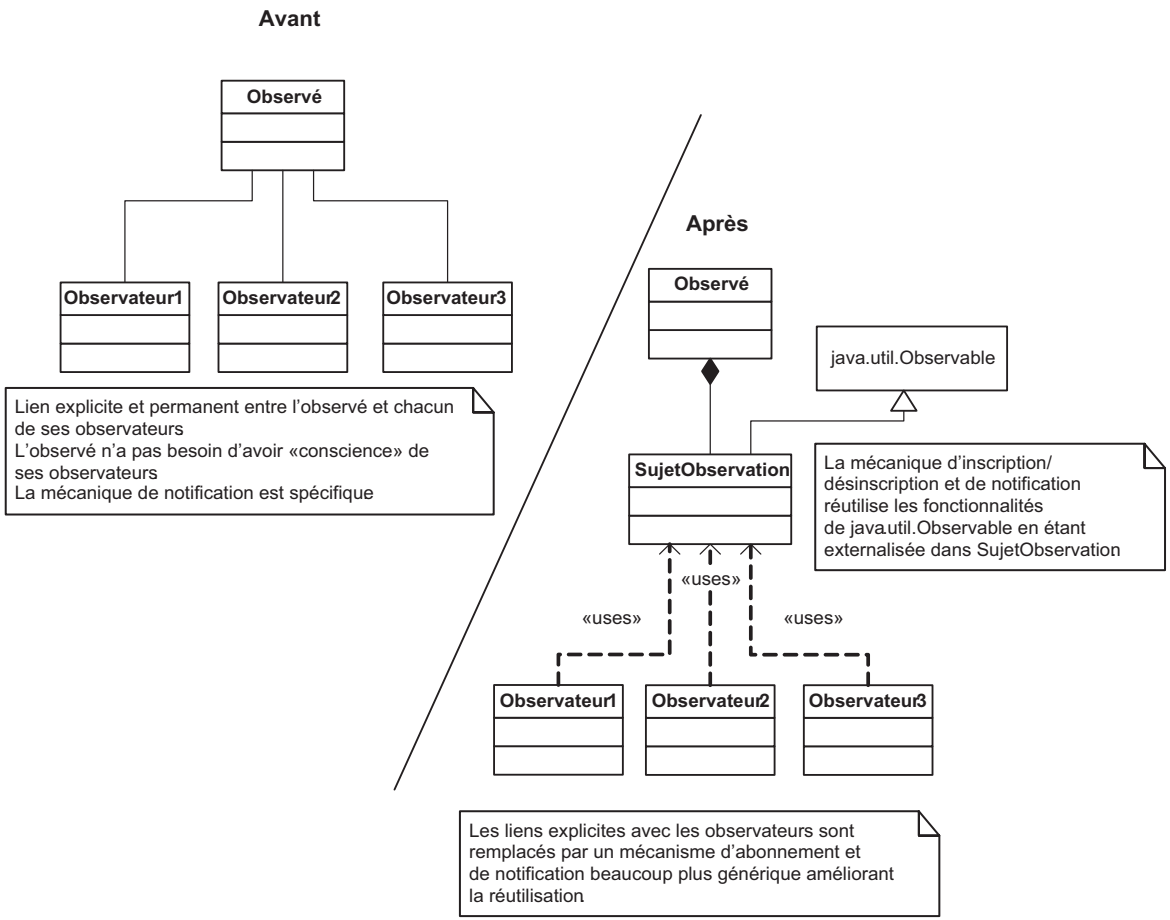

Figure 6.1

Refonte avec le design pattern observateur

Gains attendus et risques à gérer

Le gain attendu par l'utilisation du design pattern observateur est de simplifier les relations unidirectionnelles (observé vers ses observateurs) qu'entretient une classe donnée avec les autres. Grâce à ce design pattern, nous pouvons convertir des relations « en dur » (la classe connaît explicitement tous ces observateurs, créant ainsi une dépendance directe au niveau du code) par un processus générique et dynamique beaucoup plus souple à maintenir et à faire évoluer.

Les risques à gérer sont proportionnels au nombre d'observateurs identifiés pour une classe donnée puisque la mise en place de ce design pattern les impacte directement. Il faut particulièrement veiller à ce que les relations entre les observateurs et l'observé soient unidirectionnelles (observé vers observateurs). Le fonctionnement de la classe ne doit pas être « perturbé » par le fait qu'elle est observée.

Il faut toutefois veiller à ne pas abuser de ce design pattern, applicable potentiellement à un grand nombre de situations, et à le réserver à des cas où sa généricité constitue un apport réel par rapport à l'existant.

Moyens de détection des cas d'application

Les classes ayant un couplage afférent important peuvent cacher des sujets d'observation. Cependant, la métrique du couplage est insuffisante pour détecter efficacement la plupart des cas d'utilisation dans lesquels une classe est observée par un faible nombre d'observateurs.

Seule une étude des interactions entre les classes permet de connaître réellement les bons candidats. Afin de réduire le périmètre de recherche, il est préférable de se concentrer sur les classes importantes du logiciel, que ce soit par leur complexité ou leur criticité.

Modalités d'application et tests associés

La première étape consiste à déterminer les sujets d'observation et les observateurs. Chaque sujet d'observation et ses changements d'état doivent être clairement identifiables afin d'extraire la logique correspondante et de la réinjecter dans la mécanique de notification. Pour chaque observateur, il faut s'assurer qu'il n'entretient pas d'autres relations avec le sujet d'observation ne pouvant être couvertes par le mécanisme de notification.

Afin de minimiser les risques, nous conseillons de mettre en place ce design pattern de manière progressive, c'est-à-dire un sujet d'observation à la fois, observateur par observateur.

Chaque sujet d'observation nécessite d'avoir une classe interne héritant de la classe `java.util.Observable`. Cette classe interne doit surcharger les méthodes `notifyObservers` pour pouvoir déclencher la notification par la classe observée. En effet, cette dernière étant la mieux placée pour signaler un changement d'état d'un sujet d'observation, c'est elle qui appelle les méthodes `notifiyObservers`.

Pour chacune des classes internes, un attribut de ce type dans la classe observée est créé. C'est lui qui est utilisé pour déclencher les notifications et gérer les abonnements des observateurs. Concernant ce dernier point, il est nécessaire de le rendre accessible aux observateurs afin qu'ils puissent s'inscrire. Pour cela, une méthode `get` doit être créée spécifiquement pour cet attribut.

Une fois la mécanique d'inscription/désinscription et de notification en place dans la classe observée, il faut la tester. Pour cela, des tests unitaires avec des simulacres d'objets pour les observateurs doivent être mis en place pour chaque sujet d'observation et chaque changement d'état associé.

Ensuite, chaque observateur doit être modifié de manière qu'il s'inscrive auprès du ou des sujets d'observation qui l'intéressent (un même observateur peut observer plusieurs classes et plusieurs sujets d'observation). Pour cela, il utilise la méthode `get` précédemment créée pour le sujet d'observation qui l'intéresse afin de pouvoir appeler la méthode `addObserver`.

Pour terminer, chaque observateur doit implémenter l'interface `java.util.Observer` en implémentant sa méthode publique `update`. Cette méthode doit contenir la logique déclenchant les traitements idoines lors d'un changement d'état d'un sujet d'observation (la méthode `update` est utilisée pour tous les sujets d'observation, la différentiation se faisant grâce à son premier paramètre, qui renvoie la référence à l'objet `Observable` ayant déclenché la notification).

Pour chaque observateur, il est important de tester le bon traitement des notifications. Nous pouvons écrire des tests unitaires avec, si nécessaire, un simulacre d'objet pour le sujet d'observation. Une fois l'observateur testé, les liens directs entre le sujet d'observation et ses observateurs peuvent être supprimés au profit des mécanismes du design pattern observateur.

Il est alors nécessaire de tester l'ensemble à partir du sujet d'observation afin de s'assurer qu'il n'y a pas eu de régression. Pour cela, nous pouvons utiliser des tests validés sur la version originelle de la classe observée.

Exemple de mise en œuvre

Supposons que, dans le cadre d'un site Web marchand, une classe `Panier` soit définie pour traiter le panier d'achat des clients. Cette classe comporte une méthode `declencheCommande`, qui déclenche les traitements associés à la validation de la commande par le client (vérification des coordonnées, paiement, contrôle des stocks, etc.). Ces traitements obligent la classe `Panier` à avoir un accès à différents objets du logiciel pour les prendre en charge, notamment les objets de gestion de stock et de comptabilité.

Une implémentation possible, partiellement reproduite ici, de la classe `Panier` pourrait se présenter de la manière suivante :

```
package fr.eyrolles.exemples.patterns.observateur;

import java.util.Vector;

public class Panier {
    //...
    private GestionDeStock stock;
    private Comptabilite compta;
    private Vector contenu;
    //...

    public void declencheCommande() {
       //...
       stock.traite(contenu);
       compta.traite(contenu);
       //...
    }
}
```

Fondamentalement, la classe `Panier` n'a pas besoin d'avoir un lien direct avec la gestion de stock et la comptabilité. La validation d'une commande par le client doit être vue comme un événement déclenchant plusieurs traitements indépendants, dont le détail n'a

pas besoin d'être connu par la classe `Panier`. L'utilisation du design pattern observateur a donc tout son sens dans cette situation.

Dans cet exemple, le sujet d'observation est le déclenchement de la commande. Nous allons créer une classe interne dérivant de `java.util.Observable`, l'instance correspondante au sein du sujet d'observation, et la méthode `get` permettant d'accéder à l'attribut stockant cette instance :

```
package fr.eyrolles.exemples.patterns.observateur;
import java.util.Vector;
import java.util.Observable;

public class Panier {
   //...
   private DeclenchementCommande sujet =
     new DeclenchementCommande();

   public DeclenchementCommande getSujet() {
     return sujet;
  }

  //...
  public class DeclenchementCommande extends Observable {
     public void notifyObservers() {
       super.setChanged();
       super.notifyObservers();
     }

     public void notifyObservers(Object p) {
       super.setChanged();
       super.notifyObservers(p);
     }
  }
}
```

Nous constatons dans l'implémentation de la classe interne que des appels à la méthode `setChanged` sont effectués. Ils sont nécessaires pour indiquer qu'il y a eu changement d'état du sujet d'observation. Sans ces appels, les appels aux méthodes `notifyObservers` qui suivent sont sans effet, et les observateurs ne sont pas notifiés.

Pour terminer avec la classe `Panier`, il est nécessaire de mettre en place la mécanique de déclenchement des notifications. Pour cela, il suffit de modifier la méthode `declencheCommande` pour qu'elle appelle la méthode `notifyObservers` de l'attribut `sujet` :

```
public void declencheCommande() {
    sujet.notifyObservers(contenu);
    //...
    stock.traite(contenu);
    compta.traite(contenu);
    //...
}
```

Après avoir testé le bon fonctionnement de cette nouvelle mécanique, nous pouvons modifier les classes `GestionDeStock` et `Comptabilite`. Pour `GestionDeStock`, nous devons implémenter l'interface `Observer` (la logique est similaire pour `Comptabilite`) :

```
package fr.eyrolles.exemples.patterns.observateur;

import java.util.Observable;
import java.util.Observer;
import java.util.Vector;

public class GestionDeStock implements Observer {
   //...

   public void traite(Vector p) {
       //...
   }

   public void update(Observable pSujet, Object pArg) {
      if (pSujet instanceof Panier.DeclenchementCommande) {
          traite((Vector)pArg);
      }
   }
}
```

Ici, la méthode `update` est prévue pour traiter différents sujets d'observation. La différentiation des sujets s'effectue en testant la classe du sujet.

Maintenant que l'interface `Observer` est implémentée par ces deux classes, nous pouvons tester la notification en appelant directement la méthode `update`.

Pour achever la mise en place du design pattern observateur, il est nécessaire de mettre en place la mécanique d'inscription des observateurs. Dans cet exemple, cette mécanique peut être mise en place, par exemple, par la classe chargée de la création des paniers.

Une fois cette opération terminée, nous pouvons supprimer les liens directs entre `Panier` et ses observateurs. La méthode `declencheCommande` est alors expurgée des appels aux méthodes de `GestionDeStock` et de `Comptabilite` :

```
public void declencheCommande() {
    sujet.notifyObservers(contenu);
    //...
    // SUPPRIME : stock.traite(contenu);
    // SUPPRIME : compta.traite(contenu);
    //...
}
```

Grâce au design pattern observateur, la classe `Panier` devient plus facilement réutilisable, puisque les liens directs avec la comptabilité et la gestion de stock ont disparu, et offre plus de flexibilité grâce à son mécanisme d'observation générique.

Le pattern état

Le comportement des méthodes d'une classe peut varier en fonction de l'état interne de l'objet. Si les variations sont importantes et clairement associées aux changements d'état, la complexité introduite par la gestion de ces différences en fonction de l'état peut rendre la classe difficile à maintenir.

L'idée sous-jacente du design pattern état est simple : il faut procéder par dichotomie, c'est-à-dire dissocier la gestion de l'état de l'objet de son comportement. Concrètement, cela consiste à créer une classe par état possible et de lui faire implémenter le comportement associé.

Chaque classe d'état dérive d'une même classe mère abstraite ou d'une interface spécifiant les méthodes à implémenter. Pour la classe originelle, il suffit d'instancier la classe correspondant à son état et d'utiliser ses services au travers de ses méthodes publiques. Les services sont normalisés d'un état à l'autre grâce à la classe mère abstraite ou à l'interface. La classe originelle n'a dès lors pas à se préoccuper de l'état dans laquelle elle se trouve.

Le schéma UML de la figure 6.2 illustre l'utilisation de ce design pattern dans le cadre d'une refonte.

Gains attendus et risques à gérer

Le gain attendu est la réduction de la complexité de la classe originelle et la clarification de sa structure en faisant apparaître explicitement les différents états dans lesquels un objet de ce type peut se trouver. La séparation claire des différents états facilite grandement la maintenance, et les effets de bord en cas de modification sont diminués.

La refonte d'une classe existante avec ce design pattern est cependant une tâche délicate, car elle s'effectue en profondeur. Les risques de régression sont d'autant plus importants que la gestion des états est souvent profondément enfouie dans le code de la classe existante.

Moyens de détection des cas d'application

La complexité cyclomatique peut nous aider à détecter des cas d'application de ce design pattern. Quand la gestion des états est enfouie dans les méthodes d'une classe, cela introduit souvent un grand nombre de tests pour savoir dans quel état se trouve la classe.

La gestion d'états est loin d'être la seule cause possible d'une complexité cyclomatique importante. Une analyse de la classe est nécessaire pour identifier l'applicabilité du design pattern état. Pour que celui-ci soit efficace, il est important de s'assurer que les états sont clairement définis (par exemple, marche/arrêt) et en nombre limité, que les comportements associés diffèrent fortement d'un état à l'autre et que la délégation de comportement à une classe d'état n'engendre pas d'interactions complexes entre cette dernière et la classe originelle.

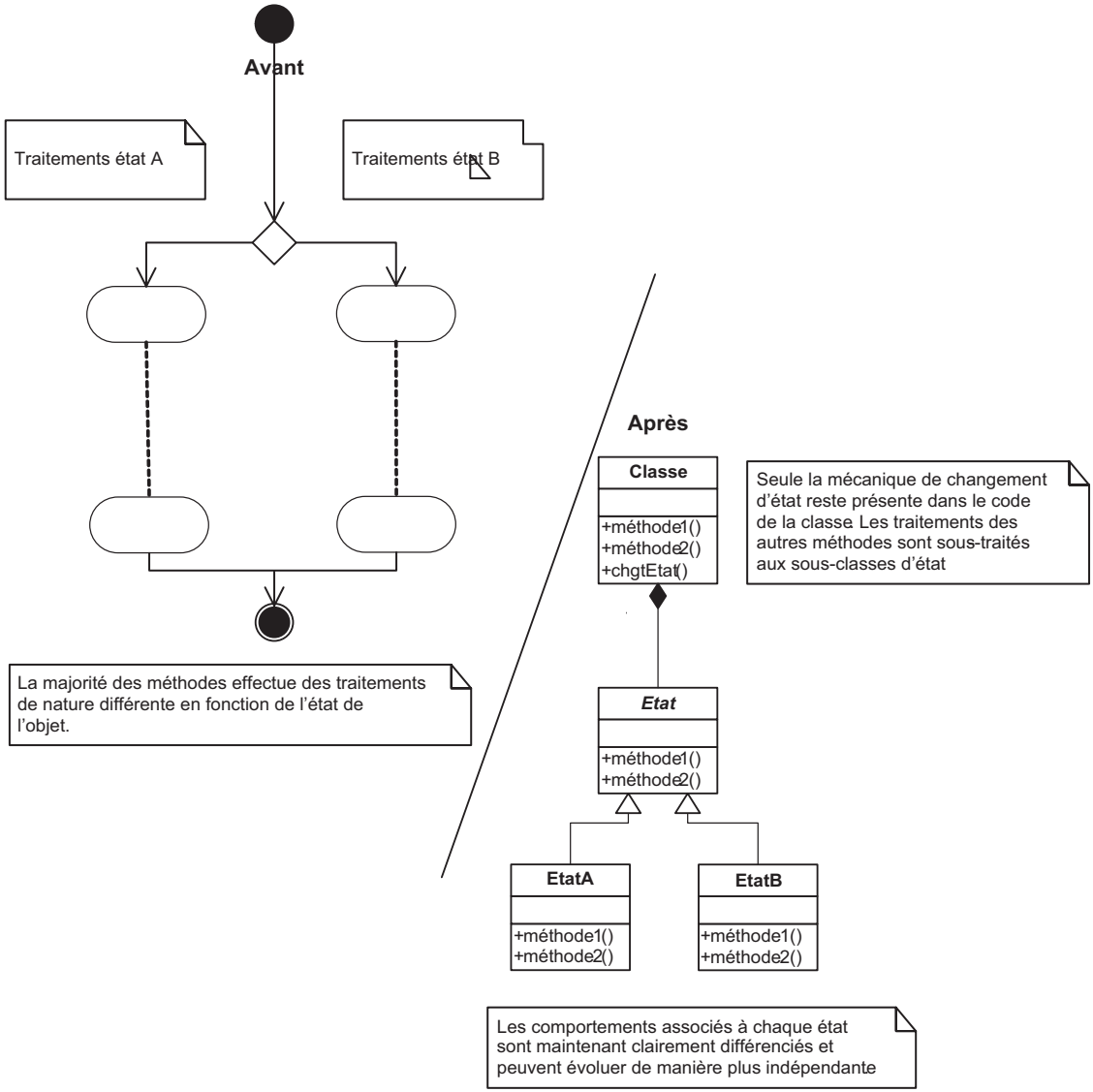

Figure 6.2

Refonte avec le design pattern état

Modalités d'application et tests associés

La première étape pour mettre en place ce design pattern est d'identifier les états de l'objet et le comportement des méthodes de la classe correspondante. Les états doivent être clairement identifiables et non ambigus (l'objet n'a qu'un seul état à l'instant *t*). Il en

va de même pour les comportements correspondants. La mécanique de transition d'un état à l'autre doit être précisément définie.

Avant d'effectuer la refonte, il est nécessaire de développer un jeu complet de tests à valider sur le code existant, s'il n'existe pas. Ce design pattern doit être totalement transparent vis-à-vis de l'extérieur. La réutilisation de ce jeu de tests est particulièrement efficace pour détecter des régressions dans la classe refondue. Il importe donc de veiller à tester chaque état identifié ainsi que toutes les transitions possibles d'un état à l'autre.

Pour matérialiser les différents états, nous utilisons des classes internes privées afin de masquer au reste du logiciel les détails d'implémentation. Cette façon de procéder facilite aussi la communication entre les états et la classe originelle, ces derniers ayant accès aux éléments privés et protégés de celle-ci (méthodes, attributs).

Chaque état dérive d'une classe abstraite ou implémente une interface reprenant l'ensemble des méthodes de la classe originelle dont le fonctionnement dépend de l'état. Ces méthodes sont bien entendu abstraites. Les méthodes de la classe originelle dont le comportement est invariant en fonction de l'état ne sont pas prises en compte.

Le choix entre une classe abstraite ou une interface dépend du contexte du logiciel. Nous préférons utiliser une interface, qui est moins lourde qu'une classe abstraite. La classe abstraite peut cependant s'avérer utile pour éviter la duplication de code entre les différents états.

Un attribut du type de la classe abstraite ou de l'interface doit être ajouté à la classe originelle. Il représente l'état courant et sera modifié par la mécanique de transition d'état.

Le contenu de chaque état est obtenu en procédant à des extractions de code au sein de la classe originelle. Pour cela, nous pouvons nous aider des techniques de base du refactoring proposées par l'environnement de développement.

Le corps des méthodes de la classe originelle dont le comportement dépend de l'état courant doit être remplacé par des appels aux méthodes correspondantes de l'attribut représentant l'état courant.

Enfin, la mécanique de changement d'état doit être mise en place. Cette mécanique dépendant très fortement du contexte, il n'y a pas de règle d'or pour son implémentation.

La refonte achevée, nous pouvons utiliser le jeu de tests mis au point sur la classe originelle pour valider la classe refondue.

Exemple de mise en œuvre

Supposons que nous ayons développé un logiciel nomade pouvant fonctionner en mode connecté à un réseau ou en mode déconnecté, sur un ordinateur portable d'un représentant de commerce, par exemple. Dans le cas du mode déconnecté, seul un sous-ensemble des fonctionnalités est disponible puisque seules les ressources de l'ordinateur sont disponibles, en l'occurrence une base de données installée sur le poste nomade.

Au niveau des classes du logiciel, cela se traduit par un comportement différent en fonction de la présence ou non d'une connexion au réseau.

Pour la classe représentant un contrat, nous avons le code suivant :

```
package fr.eyrolles.exemples.patterns.etat;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.Date;

public class Contrat {

    private ConnexionInternet connexionInternet;
    private int numero;
    private Date dateSignature;
    //...

    public Contrat chercheContrat(int pNumero)
     throws ContratInexistantException, ContratNonPresentException {
         //...
         if (connexionInternet.estDisponible()) {
             connexionBDD = BDD.getConnexion(BDD.DISTANTE);
         } else {
             connexionBDD = BDD.getConnexion(BDD.LOCALE);
         }

         requete = connexionBDD.createStatement();
         resultat = requete.executeQuery(
                   "select * from contrats where numero="+pNumero);

         if (!resultat.next()) {
             if (connexionInternet.estDisponible()) {
                 throw new ContratInexistantException(pNumero);
             } else {
                 throw new ContratNonPresentException(pNumero);
             }
         }
         //...
    }

     public void enregistreContrat() {
         //...
         if (connexionInternet.estDisponible()) {
             connexionBDD = BDD.getConnexion(BDD.DISTANTE);
         } else {
             connexionBDD = BDD.getConnexion(BDD.LOCALE);
         }
         //...
     }

     public Date getDateSignature() {
         return dateSignature;
     }

      public void setDateSignature(Date pDateSignature) {
           dateSignature = pDateSignature;
      }
```

```
    public int getNumero() {
        return numero;
    }

    public void setNumero(int pNumero) {
        numero = pNumero;
    }
}
```

Nous pouvons constater dans cet extrait que les méthodes `rechercheContrat` et `enregistreContrat` dépendent de l'état connecté ou non de la classe `Contrat`. Si nous sommes connecté, une connexion à la base de données distante est créée, sinon c'est une connexion à la base de données locale qui est créée. La base de données locale ne contenant qu'un sous-ensemble des contrats, un échec de la recherche ne signifie pas nécessairement son inexistence (le contrat peut exister dans la base de données distante contenant l'exhaustivité des contrats). L'exception générée en cas d'échec de la recherche change en fonction de l'état.

Les getters (`getDateSignature`, `getNumero`) et les setters (`setDateSignature`, `setNumero`) ne varient pas en fonction de l'état. La transition d'un état à l'autre dépend de la valeur booléenne renvoyée par la méthode `estDisponible` de l'objet `connexionInternet`.

Les conditions pour implémenter le design pattern état sont donc remplies : la classe possède deux états distincts générant un comportement spécifique.

Nous allons l'appliquer et commencer en créant une interface reprenant les deux méthodes dépendant de l'état ainsi que l'attribut représentant l'état courant. Cette interface interne à la classe `Contrat` et l'attribut se présentent de la manière suivante :

```
//...
public class Contrat {
    private Etat etatCourant;

    //...
    private interface Etat {
     public Contrat chercheContrat(int pNumero)
      throws ContratInexistantException, ContratNonPresentException;
     public void enregistreContrat();
    }
}
```

Nous créons ensuite deux classes internes implémentant cette interface (une par état, connecté et non connecté) :

```
//...
public class Contrat {
   //...
   private class EtatConnecte implements Etat {
     public Contrat chercheContrat(int pNumero)
       throws ContratInexistantException,ContratNonPresentException {
              //...
              connexion = BDD.getConnexion(BDD.DISTANTE);

              requete = connexion.createStatement();
```

```
            resultat = requete.executeQuery(
                    "select * from contrats where numero="+pNumero);

            if (!resultat.next()) {
                throw new ContratInexistantException(pNumero);
            }
            //...
        }

        public void enregistreContrat() {
                connexion = BDD.getConnexion(BDD.DISTANTE);
                //...
        }
    }

    private class EtatDeconnecte implements Etat {
        public Contrat chercheContrat(int pNumero)
         throws ContratInexistantException,ContratNonPresentException {
                //...
                connexion = BDD.getConnexion(BDD.LOCALE);

                requete = connexion.createStatement();
                resultat = requete.executeQuery(
                      "select * from contrats where numero="+pNumero);

                if (!resultat.next()) {
                    throw new ContratInexistantException(pNumero);
                }
                //...
        }

        public void enregistreContrat() {
                connexion = BDD.getConnexion(BDD.LOCALE);
               //...
        }
    }
}
```

Nous constatons que la séparation des comportements entre les états connecté et déconnecté a introduit une duplication de code (la requête dans `chercheContrat`). Cette duplication peut être évitée en factorisant le comportement commun au sein d'une classe abstraite `Etat` en lieu et place de l'interface.

La mécanique de changement d'état peut être factorisée au sein d'une méthode privée appelée `changeEtat`. Cette mécanique doit être appelée avant chaque appel aux méthodes des états. Par ailleurs, le corps des méthodes de la classe originelle doit être remplacé par des appels aux méthodes correspondantes de l'état courant (matérialisé par l'attribut `etatCourant`) :

```
//...
public class Contrat {
   //...
   Etat etatCourant;
```

```
    private void changeEtat() {
          if (connexionInternet.estDisponible()) {
              etatCourant = new EtatConnecte();
          } else {
              etatCourant = new EtatDeconnecte();
          }
    }

    public Contrat chercheContrat(int pNumero)
      throws ContratInexistantException, ContratNonPresentException {
          changeEtat();
          etatCourant.chercheContrat(pNumero) ;
   }

   public void enregistreContrat() {
       changeEtat();
       etatCourant.enregistreContrat() ;
   }

   //...
}
```

Une autre façon d'implémenter cette mécanique consisterait à utiliser le design pattern observateur afin que la classe `Contrat` soit notifiée de la disponibilité ou de l'indisponibilité de la connexion Internet. De chaque notification résulterait un changement d'état.

Comme nous pouvons le constater, la vision que la classe `Contrat` présente à l'extérieur reste inchangée. Nous pouvons donc nous reposer sur un jeu de tests construit sur la version originelle pour détecter des régressions.

Grâce à l'application du design pattern état, les comportements connectés et déconnectés sont bien distingués, facilitant leur maintenance et leur évolution.

Le pattern interpréteur

Certains traitements récurrents nécessaires à un logiciel s'avèrent soit difficiles ou fastidieux à programmer dans le langage Java du fait de sa généralité, soit nécessitant un paramétrage complexe pour prendre en compte tous les contextes utilisateur possibles.

Face à ces problèmes, l'utilisation d'un langage plus spécialisé que Java peut nettement simplifier le logiciel. Un exemple classique est le traitement des chaînes de caractères. La classe `java.lang.String` est très pauvre pour la manipulation de chaînes de caractères, abstraction faite de certaines méthodes apparues avec l'API de J2SE 1.4 *(voir ci-après l'exemple de mise en œuvre).* Or, dans un logiciel de gestion, par exemple, le contrôle de la validité des données textuelles saisies est important. La réalisation de contrôles complexes peut s'avérer extrêmement coûteuse en développement et en test. Il est donc préférable d'utiliser un langage spécialisé, comme les expressions régulières.

Là où, en Java, un programme de quinze lignes est nécessaire, l'utilisation des expressions régulières permet de le ramener à quelques lignes, réduisant d'autant la complexité du logiciel. Java étant un langage universel, il est possible de trouver des interpréteurs

d'expressions régulières écrits en Java directement utilisables dans le code d'un logiciel. J2SE en a introduit un dans l'API `java.util.regex` à partir de la version 1.4.

Concernant les traitements ayant un paramétrage complexe pour prendre en compte tous les contextes utilisateur possibles, il peut être préférable de les rendre entièrement reprogrammables à l'aide d'un langage plus simple que Java. C'est typiquement l'idée des langages de script des progiciels de gestion (ERP), ces derniers ne pouvant intégrer toutes les spécificités possibles d'une entreprise à une autre. Là encore, Java bénéficie d'interpréteurs pour des langages de script, à l'image de Jython *(http://www.jython.org)* ou de Groovy *(http://groovy.codehaus.org),* qui peuvent interagir avec des traitements Java.

Gains attendus et risques à gérer

En utilisant des langages interprétés plus spécialisés que Java, nous cherchons à réduire la complexité du code en nous reposant sur les apports fonctionnels apportés par le nouveau langage. Bien entendu, il est nécessaire que l'interpréteur soit parfaitement intégré à l'environnement Java pour que l'opération ait un intérêt.

Par la combinaison de Java et des langages de script, dont les traitements sont modifiables sans recompilation, nous cherchons à introduire plus de flexibilité dans le logiciel et à réduire la complexité introduite par la gestion de spécificités d'utilisateurs ou d'un paramétrage complexe influençant fortement les traitements.

La mise en place d'un interpréteur n'est cependant pas une tâche facile, car elle introduit les risques spécifiques suivants, en plus de la régression :

- **Bogues de l'interpréteur.** Comme tout composant logiciel, l'interpréteur possède ses propres bogues. Le débogage de ce type de composant peut nécessiter des compétences en théorie des langages, voire s'avérer impossible si l'accès aux sources n'est pas possible.
- **Bogues dans les programmes écrits dans le nouveau langage.** Le support d'outils de débogage pour ce type de langage est généralement inexistant, ce qui complexifie la correction.
- **Performances.** L'interprétation est une opération coûteuse. Il faut donc veiller à ne pas l'utiliser dans des traitements nécessitant des performances optimales.

Afin de limiter les risques, il est préférable d'éviter les utilisations trop généralisées de ce design pattern et de concentrer son emploi sur les zones bénéficiant très fortement de ses apports.

Moyens de détection des cas d'application

Dans le cas des langages spécialisés, le mieux est de se demander quel serait leur apport dans le logiciel (approche top-down) par rapport à ses fonctionnalités. Typiquement, pour un logiciel de gestion devant effectuer de nombreux contrôles des données saisies, l'apport des expressions régulières est appréciable.

Dans le cas des langages de script, le mieux est d'étudier le paramétrage du logiciel. Si celui-ci est complexe et influe fortement sur certains traitements, il peut être intéressant de transformer ces derniers en scripts. Par ailleurs, si le logiciel dispose de très nombreuses variantes, dont les différences sont concentrées sur quelques traitements, il peut être intéressant de les fusionner en une version unique, ne variant qu'au travers du contenu des scripts qui l'accompagnent.

Modalités d'application et tests associés

La première étape consiste à cartographier les traitements pouvant bénéficier des services de l'interpréteur. Cette cartographie sert de base pour évaluer les gains et les risques à mettre en place ce design pattern. Il s'avère particulièrement efficace si les traitements concernés subissent de nombreuses opérations de maintenance.

Comme il s'agit de transformer des traitements existants dans un nouveau langage, nous sommes face à une réécriture. L'effort de test pour détecter les régressions est donc important. Un jeu de tests le plus complet possible doit être mis en place sur le code originel avant de procéder à la refonte. L'effort de test doit être d'autant plus important que l'utilisation d'un nouveau langage, mal maîtrisé par les développeurs, introduit des risques supplémentaires.

La migration des traitements existants vers le nouveau langage se fait souvent manuellement. Si les traitements sont nombreux, il peut être intéressant d'évaluer des solutions de transformation automatique. Leur coût d'automatisation peut être compensé par des gains de productivité sur de gros volumes de code.

Une fois la migration effectuée, les tests mis en place sur la version originelle peuvent être réutilisés. Nous considérons que l'introduction du design pattern interpréteur ne modifie que l'implémentation des classes concernées.

Exemple de mise en œuvre

Dans les applications Web, la gestion des formulaires constitue une part importante des traitements liés à l'interface homme-machine (IHM). Dans cette gestion, le contrôle des valeurs saisies dans les champs des formulaires est souvent important et nécessaire, surtout du point de vue de la sécurité, pour éviter l'injection SQL, par exemple.

En utilisant les fonctions du langage Java, ce contrôle est particulièrement fastidieux et se matérialise souvent par une succession de conditions assez longues et sans grande valeur ajoutée.

Par exemple, le code suivant teste qu'un champ `date` est au format attendu par le logiciel (jj/mm/aaaa). Il s'agit d'une gestion élémentaire, qui ne tient pas compte de toutes les subtilités liées aux dates, comme les années bissextiles :

```
public boolean verifieDate(String pDate) {
    int jour;
    int mois;
    int annee;

    if (!(pDate.length()==10)) {
        return false;
    }
    try {
        jour = Integer.parseInt(pDate.substring(0,2));
        mois = Integer.parseInt(pDate.substring(3,5));
        annee = Integer.parseInt(pDate.substring(6));
    }
    catch (NumberFormatException e) {
        return false;
    }
    if ((!pDate.substring(2,3).equals("/"))||
       (!pDate.substring(5,6).equals("/"))) {
        return false;
    }
    if ((jour>31)||(jour<1)) {
        return false;
    }
    if ((mois>12)||(mois<1)) {
        return false;
    }
    return true;
}
```

Le format jj/mm/aaaa peut être spécifié sous forme d'expression régulière de la manière suivante : `[0-3][0-9]/[0-1][0-9]/[0-9]{4}+`.

Les valeurs entre crochets représentent l'intervalle numérique autorisé pour un caractère. `[0-3]` indique que le caractère doit être compris entre 0 et 3. Comme cette sous-expression est au début de l'expression régulière, elle s'applique au premier caractère, la sous-expression suivante (`[0-9]`) au second, etc. La sous-expression `{4}+` signifie que la sous-expression qui la précède, en l'occurrence `[0-9]`, est valable pour exactement quatre caractères consécutifs.

Pour connaître toutes les possibilités des expressions régulières, reportez-vous à la javadoc de la classe `java.util.regex.Pattern`.

Le code devient le suivant :

```
public boolean verifieDate(String pDate) {
    int jour;
    int mois;
    int annee;

    if (pDate.matches("[0-3][0-9]/[0-1][0-9]/[0-9]{4}+")) {
        return false;
    }
    jour = Integer.parseInt(pDate.substring(0,2));
    mois = Integer.parseInt(pDate.substring(3,5));
    annee = Integer.parseInt(pDate.substring(6));
    if ((jour>31)||(jour<1)) {
        return false;
    }
    if ((mois>12)||(mois<1)) {
        return false;
    }
    return true;
}
```

Grâce à l'utilisation des expressions régulières, le code est réduit de neuf lignes (19 contre 28 auparavant, soit une réduction d'un tiers). Par contre, les performances sont inférieures, la version avec interpréteur étant quatre fois plus lente que la version originelle d'après nos mesures. Il faut donc veiller à n'utiliser ce design pattern que dans des cas où la performance n'est pas une priorité majeure, c'est-à-dire quand le coût de la sous-performance est supérieur aux gains en maintenance.

Le pattern stratégie

Au sein d'une classe, le traitement effectué par une ou plusieurs méthodes données peut fortement varier en fonction du contexte. Cela a pour conséquence d'obscurcir le contenu de la méthode en mêlant traitement et tests pour identifier le contexte dans lequel l'objet se situe.

À partir du moment où les variantes du traitement sont bien distinctes et en nombre limité, nous pouvons isoler chacune d'elles dans une classe spécifique, appelée une stratégie. Ces stratégies dérivent toutes d'une classe abstraite ou d'une interface spécifiant les méthodes publiques offertes par chacune des stratégies concrètes. Au niveau de la ou des méthodes dont sont extraites les stratégies, seul subsiste le traitement permettant de sélectionner la stratégie adaptée au contexte.

Le schéma UML de la figure 6.3 illustre l'utilisation de ce design pattern dans le cadre d'une refonte.

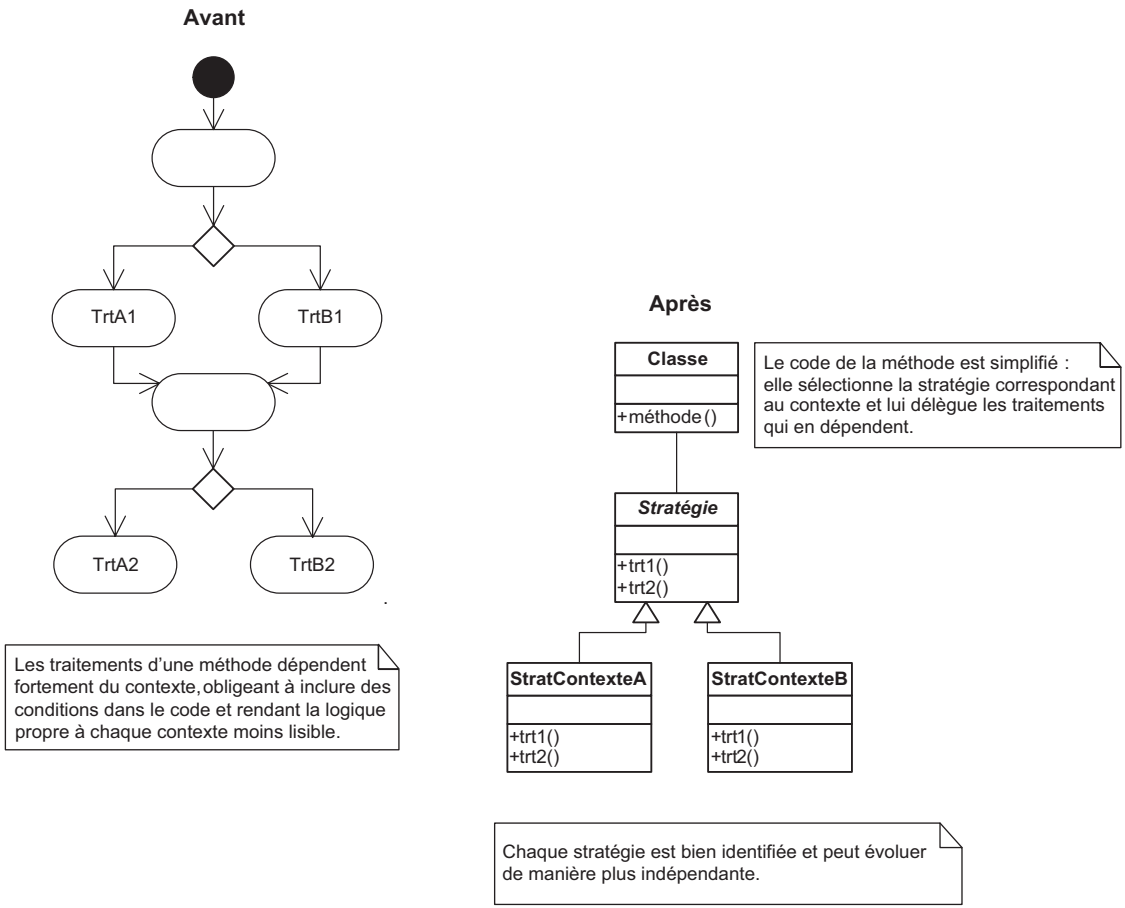

Figure 6.3

Refonte avec le design pattern stratégie

Gains attendus et risques à gérer

En dissociant traitements et gestion du contexte, nous améliorons la maintenance du code. La compréhension de stratégies spécialisées sur un contexte spécifique est généralement plus simple que celle d'un traitement adressant toutes les situations possibles. Par ailleurs, le design pattern stratégie améliore l'évolutivité du logiciel en facilitant l'introduction de nouvelles stratégies, soit en ajout, soit en remplacement des anciennes.

La complexité d'application de ce design pattern dans le cadre d'une refonte dépend du nombre de stratégies à extraire et de la complexité de leurs interactions avec le contexte. Il s'agit d'une opération délicate, sauf si l'association du contexte et de la stratégie correspondante est simple, sans ambiguïté et portant sur des traitements peu complexes. Cependant, comme il s'agit d'une opération interne à une classe, son contrat vis-à-vis de l'extérieur reste inchangé, et les tests de non-régression en sont d'autant facilités.

Lors de l'extraction des stratégies, il faut prendre garde à la duplication de code entre les différentes stratégies. Le code commun peut être factorisé au niveau de la classe abstraite dont dérivent les stratégies. Dans ce cas de figure, une interface ne peut pas être utilisée.

Moyens de détection des cas d'application

Là encore, la complexité cyclomatique peut identifier de bons candidats pour l'utilisation de ce design pattern. Cependant, du fait de la trop grande généralité de cette métrique, il est nécessaire d'analyser la méthode pour s'assurer de la pertinence de ce design pattern.

Il est notamment nécessaire de vérifier que le contenu de la méthode s'articule autour d'une succession de conditions déclenchant des traitements. Si les conditions portent de manière régulière sur les mêmes éléments, il est probable qu'ils constituent un contexte. Si la part des traitements externes à l'une de ces conditions est faible, le design pattern stratégie a de fortes chances d'être applicable.

Modalités d'application et tests associés

La première étape consiste à identifier les différents contextes influençant le contenu des traitements à effectuer. Les contextes doivent être en nombre fini et indépendants les uns des autres. Pour chaque contexte possible, les traitements associés doivent être clairement identifiés.

Avant d'effectuer la refonte, il est nécessaire de mettre au point un jeu de tests pour détecter d'éventuelles régressions. L'application de ce design pattern n'influençant que l'implémentation de la classe originelle, le jeu peut être utilisé tel quel sur le code refondu.

Une interface ou une classe abstraite, représentant la notion de stratégie, doit être créée afin de définir les méthodes à implémenter pour chaque stratégie (nous l'appelons `Strategie`). Elle doit être dans la mesure du possible interne à la classe originelle, car il s'agit d'un détail d'implémentation. Si des traitements sont communs entre différentes stratégies, ils peuvent être placés dans la classe abstraite, excluant de fait l'utilisation d'une interface pour matérialiser la notion de stratégie.

Dans la méthode originelle, ou la classe originelle si la stratégie est commune à plusieurs méthodes, il est nécessaire de créer une variable de type `Strategie` contenant la référence à la stratégie à employer (nous l'appelons `strategieCourante`). Cette variable est initialisée à l'exécution de la méthode par un test pour déterminer le contexte dans lequel cette dernière s'exécute et la stratégie correspondante.

De ce fait, une classe implémentant la notion de stratégie est créée pour chaque contexte possible. Ces classes peuvent être construites en combinant l'extraction de méthodes dans la méthode originelle et le déplacement d'éléments (les méthodes extraites). Les blocs de code extraits sont remplacés par des appels aux méthodes de la variable `strategieCourante`.

Une fois la refonte effectuée, nous pouvons utiliser le jeu de tests construit précédemment pour détecter les régressions introduites par l'opération de refactoring.

Exemple de mise en œuvre

Supposons que nous devions effectuer une refonte d'un logiciel de gestion de fiches de paie. Au sein de ce logiciel, il existe un moteur de calcul des charges patronales et salariales.

Ces calculs, très nombreux mais peu complexes, sont fortement influencés par le statut de cadre ou non du salarié et par le fait que son salaire dépasse ou non le plafond de la Sécurité sociale. Nous avons donc une logique commune aux différents contextes, mais des traitements spécifiques pour chacun d'eux :

```
public class CalculateurCharges {
    //...
    public void calculeCharges (Salarie pSalarie) {
        boolean cadre = pSalarie.isCadre();
        boolean depassePlafond = false;
        if (pSalarie.getSalaireBrut()>SECU.MONTANT_PLAFOND) {
            depassePlafond = true ;
        }

        //Calcul de la cotisation vieillesse plafonnée
        if (cadre) {
            if (depassePlafond) {
                //...
            } else {
                //...

            }
        } else {
            if (depassePlafond) {
                //...
            } else {
                //...
            }
        }
        //Calcul de la cotisation maladie
        if (cadre) {
            //...
        } else {
            //...
        }
        //Etc.
    }
}
```

Comme nous pouvons le constater, les différents contextes sont bien délimités et au nombre de quatre :

- cadre sans dépassement du plafond de la Sécurité sociale ;
- cadre avec dépassement du plafond ;
- non-cadre sans dépassement du plafond ;
- non-cadre avec dépassement du plafond.

Nous créons donc une interface interne à la classe originelle reprenant les différents calculs dépendant du contexte (nous ne reprenons que ceux donnés dans l'extrait, c'est-à-dire la cotisation vieillesse plafonnée et la cotisation maladie) :

```
public class CalculateurCharges {
    //...

    private interface Strategie {
        public void calculeVieillessePlafonnee(Salarie);
        public void calculeMaladie(Salarie);
    }
}
```

Nous créons ensuite les quatre classes internes implémentant cette interface (une par contexte possible) :

```
public class CalculateurCharges {
    //...

    private class CadreSousPlafond implements Strategie {
        public void calculeVieillessePlafonnee(Salarie pSalarie) {
            //...
        }

        public void calculeMaladie(Salarie pSalarie) {
            //...
        }
    }

    private class CadreDepPlafond implements Strategie {
        public void calculeVieillessePlafonnee(Salarie pSalarie) {
            //...
        }

        public void calculeMaladie(Salarie pSalarie) {
            //...
        }
    }

    private class NonCadreSousPlafond implements Strategie {
        public void calculeVieillessePlafonnee(Salarie pSalarie) {
            //...
        }

        public void calculeMaladie(Salarie pSalarie) {
           //...
        }
    }

    private class NonCadreDepPlafond implements Strategie {
        public void calculeVieillessePlafonnee(Salarie pSalarie) {
           //...
        }
```

```
        public void calculeMaladie(Salarie pSalarie) {
           //...
        }
    }
}
```

Pour terminer, nous modifions la méthode originelle afin qu'elle sélectionne la bonne stratégie en fonction du contexte et qu'elle utilise ses services :

```
public void calculeCharges (Salarie pSalarie) {
    Strategie strategieCourante ;

    if (pSalarie.isCadre()) {
        if (pSalarie.getSalaireBrut()>SECU.MONTANT_PLAFOND) {
            strategieCourante = new CadreDepPlafond();
        } else {
            strategieCourante = new CadreSousPlafond();
        }
    } else {
        if (pSalarie.getSalaireBrut()>SECU.MONTANT_PLAFOND) {
            strategieCourante = new NonCadreDepPlafond();
        } else {
            strategieCourante = new NonCadreSousPlafond();
        }
    }

    //Calcul de la cotisation vieillesse plafonnée
    strategieCourante.calculeVieillessePlafonnee(pSalarie);

    //Calcul de la cotisation maladie
    strategieCourante.calculeMaladie(pSalarie)

    //Etc.
}
```

Grâce à l'application du design pattern stratégie, le calcul des charges tel que présenté dans la méthode `calculeCharges` est délesté des tests rendant son code difficile à lire. Chaque stratégie est clairement isolée des autres et se concentre sur les détails de sa problématique. Ainsi, les risques de confusion au cours des opérations de maintenance, comme un taux appliqué aux cadres alors qu'il s'agit de non-cadres, sont fortement réduits.

Amélioration de la structure des classes

Les modèles structuraux aident à simplifier la structure du logiciel en introduisant des modèles de composition d'objets plus clairs et efficaces. S'ils peuvent nous aider à rendre la structure du logiciel plus lisible, il s'agit néanmoins d'opérations lourdes, à manier avec précaution.

Le pattern proxy

Le protocole d'utilisation d'un objet peut s'avérer complexe par rapport aux services effectivement rendus par celui-ci pour l'une ou l'autre des raisons suivantes :

- **Objet distant.** L'utilisation de ses services présuppose l'initialisation d'une communication *via* un protocole spécifique (RMI, IIOP, SOAP, etc.).
- **Accès contrôlés à l'objet.** La sécurité du logiciel s'impose à certains objets critiques, et l'appel de leurs méthodes est conditionné par un niveau d'autorisation suffisant.
- **Objet coûteux à instancier.** Une stratégie de création des instances est nécessaire pour optimiser les performances du logiciel.

Le design pattern proxy masque la complexité d'utilisation d'un objet en présentant une interface simplifiée. Il encapsule tout ou partie du protocole d'accès à l'objet dont il prend en charge les aspects techniques. Idéalement, un proxy se présente comme un POJO (Plain Old Java Object), c'est-à-dire un objet Java élémentaire, permettant un accès simplifié à un composant.

Le schéma UML de la figure 6.4 illustre l'utilisation de ce design pattern dans le cadre de la refonte des accès à un objet distribué.

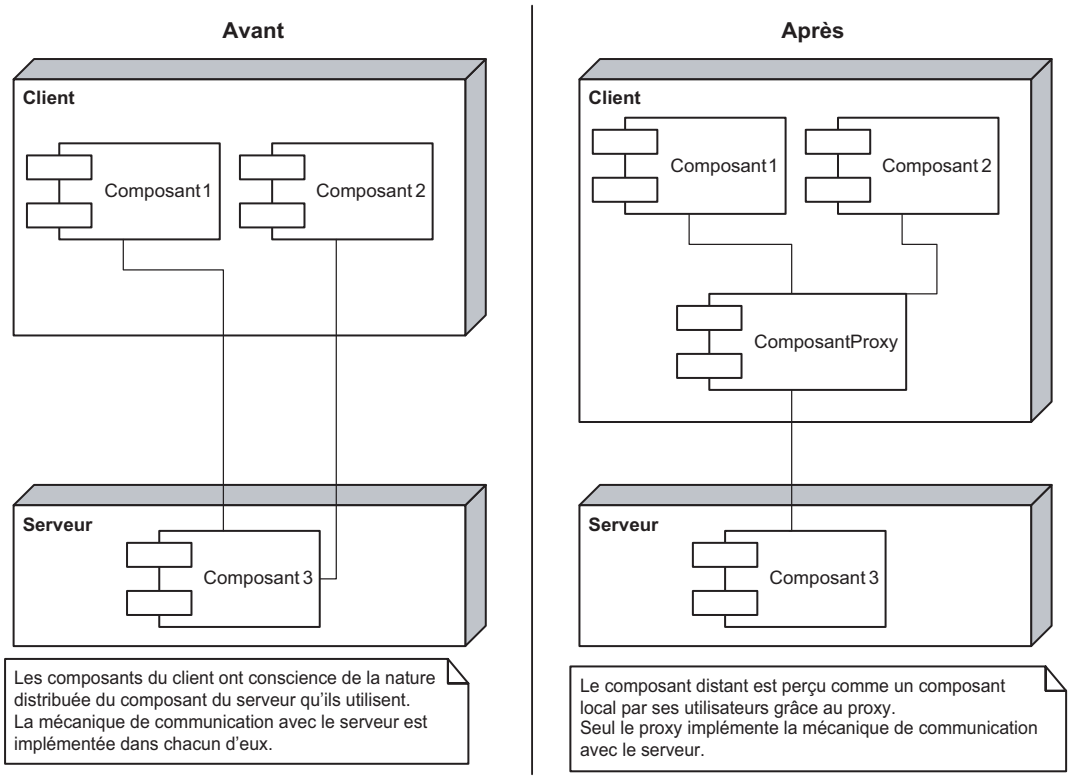

Figure 6.4

Refonte avec le design pattern proxy

Gains attendus et risques à gérer

En simplifiant le protocole d'utilisation d'un objet, le design pattern proxy facilite la compréhension de ses interactions avec ses utilisateurs. Par exemple, pour les objets distribués, il masque la mécanique de communication, rendant les changements à ce niveau transparents pour les utilisateurs du proxy.

Les risques liés à l'utilisation de ce design pattern dans le cadre d'un refactoring varient en fonction de la nature du proxy. Par exemple, dans le cadre d'un proxy d'accès à un objet distribué, une grande quantité de code peut être impactée au niveau des utilisateurs de la classe masquée par le proxy. Il est donc nécessaire de réaliser une analyse d'impacts pour estimer la portée des modifications.

Moyens de détection des cas d'application

Les cas d'utilisation les plus évidents pour le design pattern proxy au sein d'un logiciel existant sont les objets distribués, aisément identifiables dans la mesure où ils dépendent d'API techniques spécifiques (RMI, EJB, etc.). L'outil de recherche de la plupart des environnements de développement permet de les trouver sans difficulté majeure.

Les autres cas d'utilisation sont beaucoup plus difficiles à détecter sans une analyse. Dans le cas du contrôle d'accès, il peut s'agir d'extraire des objets la problématique de sécurité pour l'isoler dans un proxy, découplant ainsi la gestion de la sécurité des aspects fonctionnels. Dans le cas des objets coûteux à instancier, un passage en revue des objets les plus complexes peut permettre d'identifier rapidement de bons candidats.

Modalités d'application et tests associés

La mise en place d'un proxy est relativement simple. La première étape consiste à définir l'interface du proxy. Celle-ci est un sous-ensemble, voire la totalité, des méthodes de l'objet masqué par le proxy. Pour assurer la consistance entre proxy et objet masqué, ces deux classes doivent implémenter l'interface ainsi définie.

La classe proxy est ensuite créée. Chaque méthode de l'interface doit être implémentée. Cette implémentation dépend du type de proxy recherché. Pour un proxy de contrôle d'accès, il s'agira de vérifier que l'appelant a bien les autorisations nécessaires avant d'appeler la méthode correspondante dans l'objet masqué, etc. Une fois le proxy développé, il faut le tester unitairement en simulant un objet utilisant ses services.

Pour terminer, nous devons replacer les appels à l'objet masqué par des appels au proxy. À cette fin, une série de tests de non-régression doit être validée sur le code originel avant la modification des appels. Une fois la modification effectuée, la série de tests est rejouée pour détecter d'éventuels problèmes.

Exemple de mise en œuvre

Supposons que, dans un logiciel de gestion commerciale dont nous avons la maintenance, la sécurité soit gérée directement dans les objets métier. La classe `Contrat` aurait l'allure suivante :

```
package fr.eyrolles.exemples.patterns.proxy;

public class Contrat {
   //...

   public Contrat(int pNumero) {
     Utilisateur u = Session.getUtilisateur();
     if (GestionnaireDroits.aDroitCreation(u)) {
         // Création d'un nouveau contrat
     } else {
         throw new ViolationAccesException(u);
     }
   }
   //...

   public void supprime() {
     Utilisateur u = Session.getUtilisateur();
     if (GestionnaireDroits.aDroitSuppression(u)) {
         // suppression du contrat
     } else {
         throw new ViolationAccesException(u);
     }
   }

   public void enregistre() {
     Utilisateur u = Session.getUtilisateur();
     if (GestionnaireDroits.aDroitEcriture(u)) {
         // enregistrement
     } else {
        throw new ViolationAccesException(u);
     }
   }
}
```

Afin de bien dissocier la problématique technique de contrôle des droits du reste de la classe métier `Contrat`, il est intéressant d'employer le design pattern proxy. Le code de `Contrat` s'en trouve simplifié, et la maintenance de la sécurité peut se faire indépendamment du reste.

Après avoir mis au point un jeu de tests pour détecter d'éventuelles régressions suite à l'application du design pattern, nous pouvons créer l'interface commune au proxy et à la classe `Contrat` :

```
public interface IContrat {
   public void supprime();
   public void enregistre();
   //...
}
```

Nous créons ensuite la classe `ContratProxy`. Celle-ci consiste en une implémentation de l'interface `IContrat` à partir de l'extraction du code gérant le contrôle d'accès dans la classe `Contrat` :

```
public class ContratProxy implements IContrat {
   private Contrat contrat;

   public Contrat(int pNumero) {
     Utilisateur u = Session.getUtilisateur();
     if (GestionnaireDroits.aDroiCreation(u)) {
         contrat = new Contrat(pNumero);
     } else {
         throw new ViolationAccesException(u);
     }
   }

   //...

   public void supprime() {
     Utilisateur u = Session.getUtilisateur();
     if (GestionnaireDroits.aDroitSuppression(u)) {
         contrat.supprime();
     } else {
         throw new ViolationAccesException(u);
     }
   }

   public void enregistre() {
     Utilisateur u = Session.getUtilisateur();
     if (GestionnaireDroits.aDroitEcriture(u)) {
         contrat.enregistre();
     } else {
         throw new ViolationAccesException(u);
     }
   }
}
```

Afin d'assurer la consistance entre la classe et son proxy, il est nécessaire de supprimer tout le code de contrôle d'accès dans la classe `Contrat` et de lui faire implémenter l'interface `IContrat` :

```
package fr.eyrolles.exemples.patterns.proxy;

public class Contrat implements IContrat{
   //...

   public Contrat(int pNumero) {
     // Création d'un nouveau contrat
   }
   }

   //...

   public void supprime() {
     // suppression du contrat
   }

   public void enregistre() {
     // enregistrement
   }
}
```

Une fois cette opération terminée, le jeu de tests doit être modifié pour utiliser le proxy et non la classe `Contrat` directement. En l'exécutant, d'éventuels problèmes d'implémentation du design pattern pourront être détectés.

Pour finir, les références à la classe `Contrat` doivent être remplacées par des références à `IContrat`, séparant ainsi dans le code l'implémentation de l'interface. Les créations d'instances de `Contrat` doivent ensuite être remplacées par des créations d'instances de `ContratProxy`. Pour minimiser les risques de régression lors de cette étape, des tests devront être mis au point sur le code originel puis rejoués sur la version refondue.

Grâce au design pattern proxy, nous avons clairement séparé les problématiques de sécurité des problématiques métier de la classe `Contrat`. Leur maintenance et leurs évolutions respectives s'en trouvent d'autant facilitées.

Le pattern façade

Au sein d'un logiciel, les différents traitements pour rendre les services attendus sont répartis au sein d'une multitude d'objets. Les traitements répondent à des problématiques très variées, pouvant être fonctionnelles ou techniques. Généralement, chaque problématique est adressée par un sous-système du logiciel. Un service pouvant impliquer plusieurs problématiques, les liens entre les différents sous-systèmes peuvent devenir très nombreux et difficilement gérables.

Le design pattern façade vise à simplifier l'accès à un ensemble d'objets formant un sous-système en fournissant à l'extérieur une interface unifiée, sous forme d'une ou de plusieurs classes, masquant la complexité liée à la manipulation de cet ensemble.

Ce design pattern est particulièrement utile pour marquer clairement les limites de chaque sous-système. Typiquement, nous l'utilisons pour isoler les différentes couches techniques et fonctionnelles du logiciel. Pour que le sous-système soit correctement isolé, les dépendances doivent être unidirectionnelles, de l'extérieur vers la façade.

Le schéma UML de la figure 6.5 illustre l'utilisation de ce design pattern dans le cadre d'une refonte.

Gains attendus et risques à gérer

En améliorant la séparation des sous-systèmes entre eux, nous facilitons la compréhension de l'organisation du logiciel et la maintenance de chaque sous-système puisque la façade masque les détails de l'implémentation de ce dernier.

L'application de ce design pattern est cependant une opération lourde, car les relations entre les objets sont directement impactées. Dans le cadre d'un projet de refactoring, il est préférable d'utiliser ce design pattern de manière chirurgicale et progressive, sous-système par sous-système.

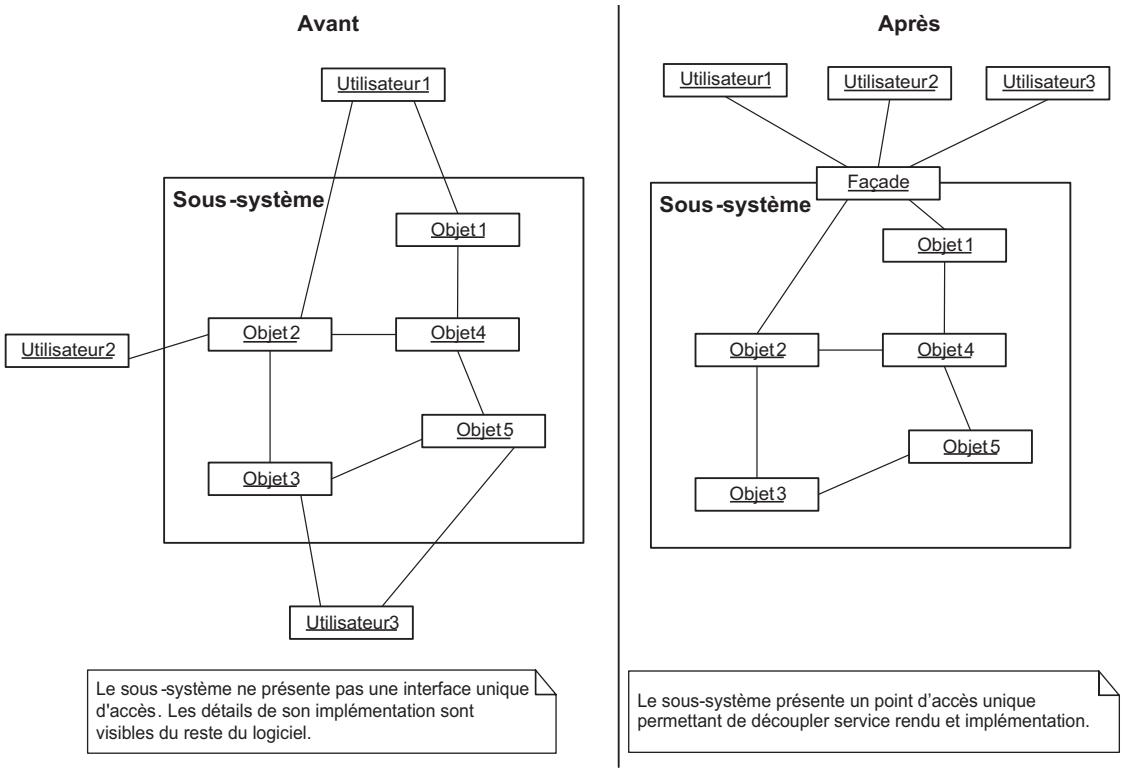

Figure 6.5

Refonte avec le design pattern façade

Il peut être utile de mettre en place des façades sur lesquelles reposeront les nouveaux développements. Celles-ci peuvent être pensées de manière à être non intrusives vis-à-vis du sous-système et autoriser deux modes de fonctionnement : soit l'accès direct aux classes du sous-système (ancien mode), soit l'accès *via* la façade (nouveau mode). Ce mode de fonctionnement implique une double maintenance et doit être considéré comme une solution palliative. La refonte de l'existant pour utiliser exclusivement la façade doit donc être planifiée.

Moyens de détection des cas d'application

Les métriques de couplage permettent d'identifier les relations trop fortes entre sous-systèmes. Bien entendu, cela suppose d'être capable de savoir quelle classe appartient à quel sous-système, ce qui n'est pas forcément évident si le logiciel a été mal conçu, notamment si les packages ne sont pas représentatifs du découpage des sous-systèmes.

Une autre méthode consiste à analyser les documents de conception ou de rétroconception, notamment les diagrammes de classes, afin d'identifier les sous-systèmes du logiciel et la nature des relations qu'ils entretiennent entre eux.

Modalités d'application et tests associés

Au préalable, il est nécessaire de délimiter le sous-système dont nous voulons masquer les détails d'implémentation. Ses interactions avec l'extérieur doivent être listées et analysées afin de les formaliser au sein de la façade.

Cette formalisation doit être accompagnée d'une série de tests permettant de valider le bon fonctionnement de la façade.

La façade peut se matérialiser sous la forme d'une ou de plusieurs classes offrant des services à l'extérieur. La façade doit masquer au maximum les détails d'implémentation du sous-système qu'elle couvre.

Ce design pattern remettant en cause les liens qui existent entre le sous-système et ses utilisateurs, nous conseillons de n'utiliser la façade qu'avec les évolutions introduites dans le logiciel. Le code existant conservant ainsi ses liens avec les composants du sous-système, nous diminuons les risques de mise en œuvre de ce design pattern très structurant pour un logiciel. Une fois son fonctionnement stabilisé avec les évolutions, son utilisation peut être généralisée à l'ensemble du logiciel.

Exemple de mise en œuvre

Reprenons la classe `Contrat` utilisée dans l'exemple de mise en œuvre du design pattern proxy. Supposons que les méthodes `supprime` et `enregistre` accèdent directement à la base de données *via* JDBC et qu'il en aille de même pour les autres objets métier, comme `Client` ou `Commercial`. Dans ce cas de figure, nous avons un lien direct entre les objets métier et le sous-système JDBC. Les problématiques techniques liées à la persistance des données sont mélangées avec les problématiques fonctionnelles adressées dans les objets métier.

Afin de simplifier le code des objets métier, il peut être intéressant d'en extraire la gestion de la persistance et de la sous-traiter à une façade chargée de la communication avec le sous-système JDBC pour assurer ce service.

Pour la classe `Contrat`, cette opération consiste à extraire le code des méthodes à supprimer et enregistrer pour les placer dans une nouvelle classe représentant la façade :

```
package fr.eyrolles.exemples.patterns.facade;

// Imports

public class FacadePersistance {
    //...
    public void supprimeContrat(Contrat pContrat) {
        //Code extrait et adapté de la méthode Contrat.supprime
    }

    public void enregistre(Contrat pContrat) {
        //Code extrait et adapté de la méthode Contrat.enregistre
    }
    //...
}
```

Afin de découpler l'implémentation, représentée par `FacadePersistance`, de son interface et faciliter les tests unitaires ainsi que les évolutions de la couche persistance, nous pouvons définir une interface `IFacadePersistance` reprenant les méthodes publiques de la classe et devant être implémentée par cette dernière :

```
package fr.eyrolles.exemples.patterns.facade;

public interface IFacadePersistance {
    public void supprimeContrat(Contrat pContrat);
    public void enregistre(Contrat pContrat);
    //...
}
```

Les références à la façade devront être du type `IFacadePersistance` au lieu de `FacadePersistance`, permettant d'avoir plusieurs stratégies de persistance, par exemple, dans le cas d'une application nomade *(voir le design pattern état).*

Une fois la façade achevée, nous devons la tester unitairement avant de l'utiliser pour les nouveaux développements.

Ensuite, pour ne pas avoir à modifier massivement le code du logiciel, les méthodes `supprime` et `enregistre` de la classe `Contrat` sont modifiées afin d'appeler la façade, au lieu d'effectuer le traitement elles-mêmes, et d'éviter une double maintenance, coûteuse et source d'erreurs. Elles doivent être marquées comme étant obsolètes (tag javadoc `@deprecated`) :

```
package fr.eyrolles.exemples.patterns.facade;

import java.util.Date;

public class Contrat {
   //...
   int numero;
   Date dateEffet;
   //...
   public int getNumero() {
     return numero;
   }

  public void setNumero(int pNumero) {
     numero = pNumero;
  }

  public Date getDateEffet() {
     return dateEffet;
  }

  public void setDateEffet(Date pDateEffet) {
     dateEffet = pDateEffet ;
  }
```

```
  /**
   * @deprecated
   */
  public void supprime() {
     facadePersistance.supprime(this);
  }

  /**
   * @deprecated
   */
  public void enregistre() {
    facadePersistance.enregistre(this);
  }

  //...
}
```

Afin de détecter d'éventuelles régressions, un jeu de tests sur les méthodes modifiées, préalablement validé sur le code originel, doit être utilisé.

Des opérations de nettoyage du code devront être planifiées par la suite pour remplacer les appels aux méthodes obsolètes par des appels directs à la façade. Chaque opération devra être accompagnée de tests de non-régression.

Une fois le code totalement nettoyé, la classe `Contrat` pourra être allégée en supprimant les méthodes obsolètes. Dans notre cas, `Contrat` devient un JavaBean élémentaire uniquement doté des méthodes getters et setters sur ses attributs :

```
package fr.eyrolles.exemples.patterns.facade;

import java.util.Date;

public class Contrat {
  //...
  int numero;
  Date dateEffet;

  //...
  public int getNumero() {
     return numero;
  }

  public void setNumero(int pNumero) {
     numero = pNumero;
  }

  public Date getDateEffet() {
     return dateEffet;
  }

  public void setDateEffet(Date pDateEffet) {
     dateEffet = pDateEffet ;
  }
}
```

Grâce à l'application du design pattern façade, les relations avec le sous-système JDBC sont centralisées et apparaissent sous forme de services métier, plus faciles à manipuler, vis-à-vis des sous-systèmes de plus haut niveau. La gestion de la persistance peut évoluer sans pour autant perturber le reste de l'application du moment que le contrat de la façade reste constant.

Le pattern adaptateur

Au cours de la vie d'un logiciel, certaines classes sont susceptibles de voir leur contrat vis-à-vis de l'extérieur profondément modifié (changement de la signature de méthodes, suppression de méthodes, etc.). Pour gérer ces modifications de contrat, deux stratégies sont possibles :

- Modifier l'ensemble des classes utilisatrices, ce qui peut s'avérer rapidement rédhibitoire.
- Chercher à rester parfaitement compatible avec les anciens contrats, d'où une complexité accrue de la classe au fil des évolutions.

Le design pattern adaptateur permet de maintenir différentes versions d'une même classe au sein d'un logiciel. Pour cela, il suffit de créer une classe par version. L'organisation de ces classes dépend du contexte. Afin de faciliter la réutilisation entre les versions et leur substitution auprès des classes utilisatrices (passage d'une version n à une version $n + 1$), nous utilisons généralement les mécanismes d'héritage : une classe mère abstraite concentrant le contrat de base commun à toutes les versions et une succession de sous-classes (une par version).

Le schéma UML de la figure 6.6 illustre l'utilisation de ce design pattern dans le cadre d'une refonte.

Gains attendus et risques à gérer

L'utilisation de ce design pattern est recommandée plutôt *a priori,* c'est-à-dire au moment où nous faisons évoluer le contrat d'une classe dans le cadre d'une maintenance évolutive, qu'*a posteriori,* lors d'un refactoring, où sa mise en œuvre est complexe. Son application peut-être justifiée dans le cas de classes rendues « obèses » et difficilement maintenables pour garantir une compatibilité ascendante complexe. Cette technique doit être vue comme un moyen de lisser le coût d'une migration à une nouvelle version et non comme une fin en soi.

Les risques associés à l'utilisation de ce design pattern *a posteriori* dans le cadre d'un refactoring sont importants, car l'implémentation de la classe originelle est modifiée en profondeur. Le risque est proportionnel au nombre de versions différentes et à l'importance de leurs différences. Par ailleurs, il faut être capable de bien faire le lien entre une version donnée et ses utilisateurs.

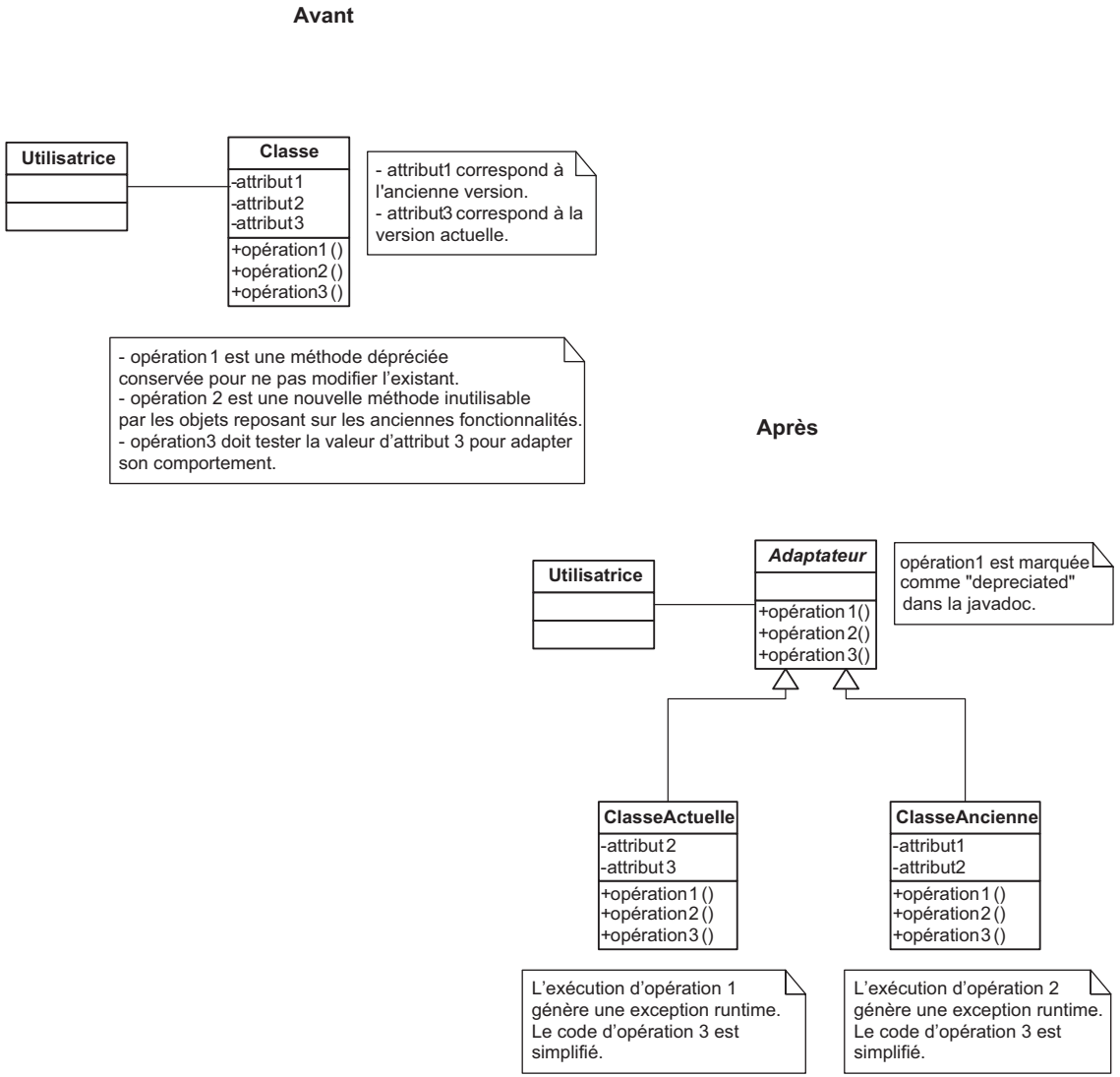

Figure 6.6

Refonte avec le design pattern adaptateur

Moyens de détection des cas d'application

Les cas d'application de ce design pattern peuvent être détectés grâce à plusieurs métriques :

- **Nombre de versions différentes d'une même classe dans l'outil de gestion de configuration.** Une classe ayant un grand nombre de versions a une probabilité non négligeable de véhiculer des traitements pour assurer la compatibilité avec le code reposant sur ses versions antérieures.

- **Classes devenues obèses** (dont les métriques de dimension sont devenues très importantes au fil du temps). Nous pouvons observer dans ce type de classe des phénomènes de sédimentation du code, dont l'évolution a été guidée par des ajouts fonctionnels non contrôlés du point de vue de la conception.
- **Classes ayant un fort couplage afférent.** Le coût de modification du contrat de ce type de classe étant très important, elles peuvent se complexifier afin de garantir une compatibilité ascendante.

Bien entendu, les métriques ne sont qu'un indice, et l'opportunité d'application du design pattern adaptateur doit être confirmée par une analyse du code de la classe sélectionnée quantitativement.

Modalités d'application et tests associés

La première étape consiste à identifier le nombre de versions différentes cohabitant au sein de la classe originelle ainsi que les méthodes implémentées pour chacune d'elles. La version en cours doit être isolée des autres afin de bien voir les différences.

Une série de jeux de tests doit être définie et validée sur la classe existante pour chaque version identifiée.

La classe originelle doit être transformée en classe abstraite. Les méthodes conservées à seule fin de compatibilité avec les versions antérieures doivent être marquées comme étant obsolètes. Pour cela, nous utilisons le tag javadoc `@deprecated`. Les développeurs sont avertis de la sorte que ces méthodes ne doivent plus être utilisées.

Ensuite, chaque version est isolée dans une classe concrète héritant de la classe originelle. Au sein de cette classe, les méthodes utilisées par cette version doivent être implémentées en extrayant le code des méthodes correspondantes dans la classe mère ainsi que les attributs nécessaires à leur exécution. *In fine,* la classe mère devient vide de toute implémentation, et ses méthodes sont déclarées abstraites, hormis pour les traitements communs à toutes les versions.

Les méthodes qui ne doivent pas être utilisées pour une version donnée, par exemple, les méthodes créées dans une version ultérieure et incompatibles avec les anciens traitements, génèrent une exception d'exécution (Runtime Exception).

Les jeux de tests précédemment créés doivent être modifiés afin d'utiliser la classe concrète correspondant à la version dont ils doivent tester la non-régression. Cette opération faite, ils doivent être exécutés pour détecter d'éventuels problèmes.

Pour terminer, il faut remplacer les créations d'instances de la classe originelle (générant des erreurs de compilation puisqu'elle est devenue abstraite) par des créations d'instances de ses sous-classes concrètes. Il est important de s'assurer que la version sélectionnée répond bien aux besoins. S'il s'avère qu'une classe utilisatrice repose sur plusieurs versions différentes, nous sélectionnons la version la plus récente et remanions le code en conséquence. Si la classe originelle est chargée dynamiquement (*via* `Class.forName`), il faut modifier la mécanique de chargement en conséquence.

Avant d'effectuer ces modifications, il est important de définir et valider des tests de non-régression sur le code originel. Ces tests seront rejoués une fois le code modifié.

Exemple de mise en œuvre

Avec le design pattern façade appliqué à la classe `Contrat`, nous avons extrait la gestion de la persistance des données de cette dernière pour la placer dans `FacadePersistance`. Pour ne pas perturber l'existant, les méthodes `supprime` et `enregistre` ont été maintenues dans la classe `Contrat`.

Afin de lever toute ambiguïté, il peut être intéressant de séparer les versions avec et sans façade. Pour cela, nous définissons en premier lieu la classe `Contrat` comme étant abstraite (les méthodes obsolètes restent marquées par le tag javadoc `@deprecated`) :

```
package fr.eyrolles.exemples.patterns.adaptateur;

public abstract class Contrat {
  //...
  int numero;
  Date dateEffet;

  //...
  public int getNumero() {
     return numero;
  }

  public void setNumero(int pNumero) {
     numero = pNumero;
  }

  public Date getDateEffet() {
     return dateEffet;
  }

  public void setDateEffet(Date pDateEffet) {
     dateEffet = pDateEffet ;
  }

  /**
   * @deprecated
   */
  public abstract void supprime();

  /**
   * @deprecated
   */
  public abstract void enregistre();

  //...
}
```

Avant d'aller plus loin, il nous faut définir et valider un jeu de tests sur le code utilisant `Contrat` afin de détecter d'éventuelles régressions.

Nous créons ensuite deux sous-classes, `ContratLeger` et `ContratLourd`. La première correspond à la version avec façade et la seconde à la version sans façade.

Le code de `ContratLourd` consiste en une extraction des méthodes obsolètes dans la classe originelle :

```
package fr.eyrolles.exemples.patterns.adaptateur;

public class ContratLourd extends Contrat {
  //...

  /**
   * @deprecated
   */
  public void supprime() {
    facadePersistance.supprime(this);
  }

  /**
   * @deprecated
   */
  public void enregistre() {
    facadePersistance.enregistre(this);
  }

  //...
}
```

`ContratLeger`, pour sa part, ne doit pas implémenter les méthodes obsolètes :

```
package fr.eyrolles.exemples.patterns.adaptateur;

public class ContratLeger extends Contrat {
  //...

  /**
   * @deprecated
   */
  public void supprime() {
    // Génération d'une exception d'exécution
    throw new MethodeObsoleteException("ContratLeger.supprime");
  }

  /**
   * @deprecated
   */
  public void enregistre() {
    throw new MethodeObsoleteException("ContratLeger.enregistre");
  }

  //...
}
```

`ContratLeger` étant réservé aux futurs développements, les créations d'instance de la classe `Contrat` originelle sont remplacées par des créations d'instances de la classe `ContratLourd`.

Il faut s'assurer par ailleurs que les créations d'instances dynamiques tiennent compte de ce changement.

Pour terminer, il faut vérifier l'application de ce design pattern en rejouant le jeu de tests mis au point sur le code existant.

Grâce à l'application du design pattern adaptateur, nous pouvons migrer progressivement le code existant reposant sur des objets métier lourds vers notre nouvelle architecture.

Conclusion

Ce chapitre a illustré la mise en œuvre de plusieurs design patterns majeurs dans le cadre du refactoring.

Ces techniques complexes remanient en profondeur une partie de la structure du logiciel. D'une manière générale, elles ne peuvent être utilisées dans le cadre d'une maintenance classique, sauf si leur périmètre reste limité.