Spring
par la pratique

Arnaud Cogoluègnes
Thierry Templier
Julien Dubois
Jean-Philippe Retaillé

avec la contribution
de Séverine Templier Roblou
et de Olivier Salvatori

ISBN : 978-2-212-12421-7

EYROLLES

6

Test des applications Spring

Les tests sont une des activités fondamentales du développement logiciel. Ce chapitre montre comment tester une application reposant sur Spring. Les types de tests abordés sont les tests unitaires et les tests d'intégration.

Par tests unitaires, nous entendons les tests portant sur un composant unique isolé du reste de l'application et de ses composants techniques (serveur d'applications, base de données, etc.). Par tests d'intégration, nous entendons les tests portant sur un ou plusieurs composants, avec les dépendances associées. Il existe bien entendu d'autres types de tests, comme les tests de performance ou les tests fonctionnels, mais Spring ne propose pas d'outil spécifique dans ces domaines.

Nous nous arrêterons de manière synthétique sur deux outils permettant de réaliser des tests unitaires, le framework JUnit et EasyMock, ce dernier permettant de réaliser des simulacres d'objets, ou *mock objects*. Ces deux outils nous fourniront l'occasion de détailler les concepts fondamentaux des tests unitaires. Nous verrons que l'utilisation de ces outils est toute naturelle pour les applications utilisant Spring, puisque le code de ces dernières ne comporte pas de dépendance à l'égard de ce framework du fait de l'inversion de contrôle. Spring propose d'ailleurs ses propres simulacres pour émuler une partie de l'API Java EE.

Pour les tests d'intégration, nous nous intéresserons aux extensions de JUnit fournies par Spring et par le framework DbUnit, spécialisé dans les tests de composants de persistance des données. Là encore, les tests s'avèrent aisés à implémenter, ces extensions prenant en charge les problématiques techniques les plus courantes.

Pourquoi écrire des tests ?

L'écriture de tests peut paraître paradoxale au premier abord : écrire un programme pour en tester un autre. De plus, l'écriture de test peut paraître consommatrice de temps, puisqu'elle correspond à l'écriture de code supplémentaire. Cependant, les tests présentent un ensemble d'avantages qui les rendent indispensables à tout développement informatique se voulant robuste et évolutif.

Les tests permettent de guider la conception. En effet, un code difficilement testable est en général un code mal conçu. Le rendre facilement testable, unitairement ou *via* un test d'intégration, est sans conteste une avancée vers un code mieux conçu.

Les tests écrits sous forme de programme résistent à l'épreuve du temps, contrairement aux tests manuels. C'est pour cette raison que les tests font complètement partie d'un projet et doivent être sauvegardés sur un dépôt de sources, tout comme le code de l'application.

Les tests sont enfin un élément essentiel dans la vie d'une application car ils lui permettront d'évoluer et de s'adapter aux nouveaux besoins. Il est rare qu'une application ne doive pas être modifiée au cours de sa vie afin de répondre à de nouveaux besoins. Sans tests unitaires, les modifications nécessaires se feront généralement dans la crainte de régressions (« casser » des fonctionnalités existantes en en rajoutant d'autres). Cette crainte ne facilite pas un raisonnement global, qui privilégierait des modifications de conception plutôt que des modifications se rapprochant du « bricolage ». Avec des tests unitaires offrant une bonne couverture, les modifications se font plus sereinement. Des modifications plus profondes sont favorisées par les tests unitaires qui permettent de mettre en évidence les régressions. Avec une telle approche, la qualité de l'application et son adaptabilité ne se dégradent pas avec le temps mais s'améliorent.

Les tests ne sont pas la panacée ; ils ne permettront pas systématiquement à une application d'être sans faille, car ils doivent être écrits avec soin. Ils constituent cependant un engrenage essentiel dans la mécanique globale de tout développement informatique.

Les tests unitaires avec JUnit

JUnit est un framework Java Open Source créé par Erich Gamma et Kent Beck. Il fournit un ensemble de fonctionnalités permettant de tester unitairement les composants d'un logiciel écrit en Java. D'autres frameworks suivant la même philosophie sont disponibles pour d'autres langages ou pour des technologies spécifiques, comme HTTP. Ils constituent la famille des frameworks xUnit.

Initialement conçu pour réaliser des tests unitaires, JUnit peut aussi être utilisé pour réaliser des tests d'intégration, comme nous le verrons plus loin dans ce chapitre.

Dans les sections suivantes, nous indiquons comment manipuler les différents éléments fournis par JUnit afin de créer des tests pour le module « core » de Tudu Lists. Dans cette application, l'ensemble des cas de tests est regroupé dans le répertoire **src/test/java**. Les différents tests sont organisés selon les classes qu'ils ciblent. Ainsi, ils reproduisent la structure de packages de Tudu Lists.

La version de JUnit étudiée ici est la 4.5. Elle privilégie l'utilisation d'annotations pour définir les tests plutôt que l'héritage d'une classe de test (et des conventions de nommage), comme sa version 3.8, encore certainement la plus utilisée. Cette approche par annotations favorise le modèle POJO puisque le framework est relativement peu intrusif (pas de nécessité d'héritage). Elle est aussi plus souple et s'avère très bien supportée par Spring. C'est pour cet ensemble de raisons que nous avons préféré l'étude de cette nouvelle version.

Les cas de test

Les cas de test sont une des notions de base de JUnit. Il s'agit de regrouper dans une entité unique, en l'occurrence une classe Java annotée, un ensemble de tests portant sur une classe de l'application.

Chaque test est matérialisé sous la forme d'une méthode sans paramètre, sans valeur de retour et portant l'annotation `org.junit.Test`. Le nom de la classe regroupant les tests d'une classe est conventionnellement celui de la classe testée suffixée par `Test` (par exemple, `TodosManagerImplTest`).

Squelette d'un cas de test

Pour introduire la notion de cas de test, nous allons utiliser la classe `Todo` définie dans le package `tudu.domain.model`. Cette classe définit deux méthodes, `compareTo` (méthode de l'interface `java.lang.Comparable`) et `equals` (héritée de `java.lang.Object`).

Pour tester ces deux méthodes, nous allons créer plusieurs instances de la classe `Todo` et effectuer des comparaisons ainsi que des tests d'égalité. Nous définissons pour cela une classe `TodoTest` ayant deux méthodes, `compareTo` et `equals` :

```
package tudu.domain.model;

import org.junit.Test;

public class TodoTest {

   (...)

   @Test
   public void compareTo() {
      (...)
   }

   @Test
   public void equals() {
      (...)
   }
}
```

Notons que la classe de test ne dépend d'aucune classe par un héritage ou d'une interface par une implémentation. De plus, les méthodes de tests portent directement le nom des méthodes testées.

La notion de fixture

Dans JUnit, la notion de *fixture,* ou contexte, correspond à un ensemble d'objets utilisés par les tests d'un cas. Typiquement, un cas est centré sur une classe précise du logiciel. Il est donc possible de définir un attribut ayant ce type et de l'utiliser dans tous les tests du cas. Il devient alors une partie du contexte. Le contexte n'est pas partagé par les tests, chacun d'eux possédant le sien, afin de leur permettre de s'exécuter indépendamment les uns des autres. Il est important de noter qu'avec JUnit, le contexte est initialisé pour chacune des méthodes d'une classe de test.

Avec JUnit, il est possible de mettre en place le contexte de plusieurs manières. La manière la plus simple consiste à annoter des méthodes avec l'annotation `org.junit.Before`. Ces méthodes doivent avoir une signature de type `public void`. Si l'initialisation du contexte alloue des ressources spécifiques, il est nécessaire de libérer ses ressources. Cela se fait en annotant des méthodes avec l'annotation `org.junit.After`. Les méthodes annotées pour la destruction du contexte ont les mêmes contraintes que son initialisation. Il est donc fréquent que toute méthode d'initialisation ait son pendant pour la destruction du contexte.

Dans JUnit 3, le cycle de vie du contexte était géré par la surcharge de deux méthodes : `setUp` et `tearDown`. Il est donc fréquent d'adopter cette nomenclature avec l'approche annotations.

La gestion du contexte peut donc se définir de la manière suivante :

```
public class TodoTest {

   @Before
   public void setUp() {
      // Création du contexte
   }

   @After
   public void tearDown() {
      // Destruction du contexte
   }

   (...)
}
```

Pour les tests de la classe `Todo`, nous pouvons créer un ensemble d'attributs de type `Todo` qui nous serviront de jeu d'essai pour nos méthodes de test :

```
public class TodoTest {

   private Todo todo1;
   private Todo todo2;
   private Todo todo3;
```

```
    @Before
    public void setUp() {
         todo1 = new Todo();
         todo1.setTodoId("01");
         todo1.setCompleted(false);
         todo1.setDescription("Description");
         todo1.setPriority(0);

         todo2 = new Todo();
         todo2.setTodoId("02");
         todo2.setCompleted(true);
         todo2.setDescription("Description");
         todo2.setPriority(0);

         todo3 = new Todo();
         todo3.setTodoId("01");
         todo3.setCompleted(false);
         todo3.setDescription("Description");
         todo3.setPriority(0);
    }

    (...)
}
```

Les assertions et l'échec

Dans JUnit, les assertions sont des méthodes permettant de comparer une valeur obtenue lors du test avec une valeur attendue. Si la comparaison est satisfaisante, le test peut se poursuivre. Dans le cas contraire, il échoue, et un message d'erreur s'affiche dans l'outil permettant d'exécuter les tests unitaires.

Les assertions peuvent être effectuées avec des appels à des méthodes statiques de la classe `org.junit.Assert`. Leur nom est préfixé par `assert`.

Pour les booléens, les assertions suivantes sont disponibles :

```
assertFalse (boolean obtenu);
assertTrue (boolean obtenu);
```

Elles testent le booléen `obtenu` sur les deux valeurs littérales possibles, faux ou vrai.

Pour les objets, les assertions suivantes sont disponibles, quel que soit leur type :

```
assertEquals (Object attendu,Object obtenu);
assertSame (Object attendu,Object obtenu);
assertNotSame (Object attendu,Object obtenu);
assertNull (Object obtenu);
assertNotNull (Object obtenu);
```

`assertEquals` teste l'égalité de deux objets tandis qu'`assertSame` teste que `attendu` et `obtenu` font référence à un seul et même objet. Par exemple, deux objets de type `java.util.Date` peuvent être égaux, c'est-à-dire contenir la même date, sans être pour autant un seul et même

objet. `assertNotSame` vérifie que deux objets sont différents. Les deux dernières assertions testent si l'objet obtenu est nul ou non.

Pour les différents types primitifs (`long`, `double`, etc.), une méthode `assertEquals` est définie, permettant de tester l'égalité entre une valeur attendue et une valeur obtenue. Dans le cas des types primitifs correspondant à des nombres réels (`double`), un paramètre supplémentaire, le delta, est nécessaire, car les comparaisons ne peuvent être tout à fait exactes du fait des arrondis.

Il existe une variante pour chaque assertion prenant une chaîne de caractères en premier paramètre (devant les autres). Cette chaîne de caractères contient le message à afficher si le test échoue au moment de son exécution. Si l'échec d'une assertion n'est pas évident, il est en effet important de préciser la raison de l'échec avec un message.

Les assertions ne permettent pas de capter tous les cas d'échec d'un test. Pour ces cas de figure, JUnit fournit la méthode `fail` sous deux variantes : une sans paramètre et une avec un paramètre, permettant de donner le message d'erreur à afficher sous forme de chaîne de caractères. L'appel à cette méthode entraîne l'arrêt immédiat du test en cours et l'affiche en erreur dans l'outil d'exécution des tests unitaires.

Les tests unitaires n'héritant d'aucune classe, ils doivent appeler les méthodes d'assertions directement, par exemple :

```
Assert.assertTrue(someBooleanValue);
```

Cette syntaxe peut être simplifiée *via* un import statique :

```
import static org.junit.Assert.*;

(...)

// dans une méthode de test :
assertTrue(someBooleanValue);
```

Si nous reprenons notre exemple `TodoTest`, il se présente désormais de la manière suivante :

```
public class TodoTest {

   (...)

   @Test
   public void compareTo() {
      // Vérifie la consistance avec la méthode equals
      // Cf. JavaDoc de l'API J2SE
      assertTrue(todo1.compareTo(todo3)==0);

      // Vérifie le respect de la spec de Comparable pour null
      // Cf. JavaDoc de l'API J2SE
      try {
         todo1.compareTo(null);
         fail();
      }
      catch(NullPointerException e){
```

```
            // OK
        }

        // todo1 n'est pas fermé donc < à todo2
        assertTrue(todo1.compareTo(todo2)<0);

        // Vérifie que l'inverse est vrai aussi
        assertTrue(todo2.compareTo(todo1)>0);
    }

    @Test
    public void equals() {
        assertEquals(todo1,todo3);
        assertFalse(todo1.equals(todo2));
    }
}
```

Exécution des tests

Une fois les cas de test et les suites de tests définis, il est nécessaire de les exécuter pour vérifier le logiciel.

Le lanceur standard de JUnit

JUnit introduit la notion de lanceur pour exécuter les tests et propose un lanceur par défaut. Il est textuel et utilise en fait une classe écrite pour JUnit 3, le motif de conception adaptateur étant utilisé pour que des tests JUnit 4 puissent être lancés.

Pour utiliser ce lanceur en mode texte dans un cas de test (ici la classe `TodoTest`), il suffit d'y ajouter une méthode `main` de la manière suivante :

```
public static void main(String[] args) {
    junit.textui.TestRunner.run(
        new JUnit4TestAdapter(TodoTest.class)
    );
}
```

La classe `junit.textui.TestRunner` est appelée en lui passant en paramètre la classe du cas de test (et non une instance) à exécuter, *via* un adaptateur pour JUnit 4. Cette méthode `main` peut être écrite soit directement dans la classe du cas de test, comme dans cet exemple, soit dans une classe spécifique.

Si nous exécutons `TodoTest`, nous obtenons le résultat suivant dans la console Java :

```
..
Time: 0,047
OK (2 tests)
```

Ce résultat indique de manière laconique que les deux tests définis dans `TodoTest` se sont bien exécutés.

Le lanceur intégré à JUnit a le mérite d'exister, mais il sert principalement à illustrer l'utilisation du framework et on lui préfère généralement d'autres lanceurs, selon les besoins.

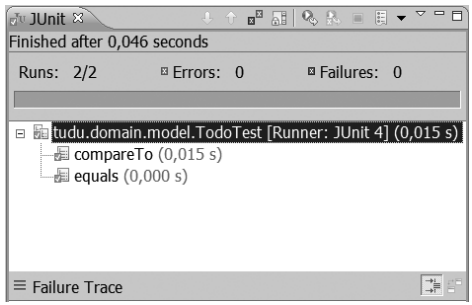
Le lanceur JUnit intégré à Eclipse

Eclipse propose son propre lanceur JUnit, parfaitement intégré à l'environnement de développement.

Pour l'utiliser, il n'est pas nécessaire de créer une méthode `main` spécifique dans les cas de test, à la différence des lanceurs standards. Il suffit de sélectionner l'explorateur de package et le fichier du cas de test par clic droit et de choisir Run dans le menu contextuel. Parmi les choix proposés par ce dernier, il suffit de sélectionner JUnit Test.

Une vue JUnit s'ouvre alors pour nous permettre de consulter le résultat de l'exécution des tests. Si tel n'est pas le cas, nous pouvons l'ouvrir en choisissant Window puis Show view et Other. Une liste hiérarchisée s'affiche, dans laquelle il suffit de sélectionner JUnit dans le dossier Java et de cliquer sur le bouton OK *(voir figure 6-1)*.

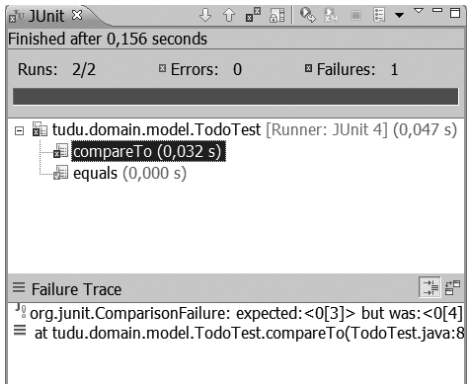
Figure 6-1
Vue JUnit intégrée à Eclipse



L'intérêt de ce lanceur réside dans sa gestion des échecs après l'exécution des tests.

Si nous modifions `TodoTest` de manière que deux tests échouent (il suffit pour cela de mettre une valeur attendue absurde, qui ne sera pas respectée par la classe testée), nous obtenons dans le lanceur le résultat illustré à la figure 6-2.

Figure 6-2
Gestion des échecs dans la vue JUnit d'Eclipse



Les échecs (*failures*) sont indiqués par une croix bleue et les succès par une marque verte. En cliquant sur un test ayant échoué, la trace d'exécution est affichée dans la zone Failure Trace. En double-cliquant sur le test, son code source est immédiatement affiché dans l'éditeur de code d'Eclipse.

À partir des résultats des tests, le développeur peut naviguer directement dans son code et le corriger dans Eclipse, ce qui n'est pas possible avec les lanceurs standards de JUnit.

Lanceurs intégrés à la construction d'un projet

Les tests unitaires font complètement partie du code source d'un projet ; il n'y a donc aucune raison pour que leur exécution ne fasse pas partie du processus de construction (« build ») d'un projet.

Les principaux outils de construction en Java (Ant, Maven) disposent donc de systèmes d'exécution des tests unitaires. La construction de Tudu Lists est faite avec Maven 2, outil qui intègre le lancement des tests unitaires *via* le plugin Surefire. Maven 2 lance tous les tests se trouvant dans un répertoire donné (généralement **src/test/java**). Le plugin Surefire est capable de lancer différents types de tests (JUnit 3 et 4, TestNG). Les résultats des tests sont stockés sous forme de fichiers (texte et XML), qui sont ensuite transformés pour générer des rapports plus lisibles, au format HTML.

La figure 6-3 illustre des rapports de tests unitaires pour la partie core de Tudu Lists.

Summary

[Summary][Package List][Test Cases]

Tests	Errors	Failures	Skipped	Success Rate	Time
125	0	0	0	100%	13.283

Note: failures are anticipated and checked for with assertions while errors are unanticipated.

Package List

[Summary][Package List][Test Cases]

Package	Tests	Errors	Failures	Skipped	Success Rate	Time
tudu.aspects.notify.classic	10	0	0	0	100%	0.718
tudu.domain.model	2	0	0	0	100%	0.015
tudu.domain.dao.ibatis	4	0	0	0	100%	0.609
tudu.service.notify.impl	1	0	0	0	100%	0.031
tudu.domain.dao.jpa	27	0	0	0	100%	1.578
tudu.integration	7	0	0	0	100%	4.485
tudu.aspects.notify.aspectj	30	0	0	0	100%	3.441
tudu.aspects.observer.aspectj	1	0	0	0	100%	0.672
tudu.domain.dao.hibernate	8	0	0	0	100%	0.844
tudu.domain.dao.jdbc	1	0	0	0	100%	0.454
tudu.service.impl	24	0	0	0	100%	0.171
tudu.security	1	0	0	0	100%	0
tudu.aspects.targetsource	2	0	0	0	100%	0.218
tudu.domain.model.comparator	7	0	0	0	100%	0.047

Figure 6-3

Rapports de tests unitaires avec Maven 2

Quel que soit le système utilisé, il est essentiel d'incorporer le lancement des tests unitaires au processus de construction et d'avoir un moyen de communiquer les résultats, afin d'avoir une idée précise de la santé du projet.

En résumé

L'écriture de tests avec JUnit n'est pas intrinsèquement compliquée. Cependant, il faut garder à l'esprit que cette simplicité d'utilisation dépend fortement de la façon dont l'application est conçue. La mise en œuvre de tests unitaires dans une application mal conçue et qui, par exemple, ne montre aucune structuration peut s'avérer très difficile, voire parfaitement impossible.

L'utilisation de JUnit est rendue efficace grâce aux principes architecturaux que doivent appliquer les applications Spring, à savoir l'inversion de contrôle apportée par le conteneur léger et la séparation claire des couches de l'application. Grâce à ces principes, les composants se révèlent plus simples à tester, car ils se concentrent dans la mesure du possible sur une seule préoccupation.

En complément de JUnit, il est nécessaire d'utiliser des simulacres d'objets afin d'isoler des contingences extérieures le composant à tester unitairement. C'est ce que nous allons voir à la section suivante.

Les simulacres d'objets

Le framework JUnit constitue un socle pour réaliser des tests, mais il n'offre aucune fonctionnalité spécifique pour tester les relations entre les différents objets manipulés lors d'un test. Il est de ce fait difficile d'isoler les dysfonctionnements de l'objet testé de ceux qu'il manipule, ce qui n'est pas souhaitable lorsque nous réalisons des tests unitaires. L'idée est en effet de tester seulement une classe et de ne pas dépendre des classes qu'elles utilisent.

Afin de combler ce vide, il est possible d'utiliser des simulacres d'objets (*mock objects*). Comme leur nom l'indique, ces simulacres simulent le comportement d'objets réels. Il leur suffit pour cela d'hériter de la classe ou de l'interface de l'objet réel et de surcharger chaque méthode publique utile au test. Le comportement de ces méthodes est ainsi redéfini selon un scénario conçu spécifiquement pour le test unitaire et donc parfaitement maîtrisé.

Par exemple, si nous testons un objet faisant appel à un DAO, nous pouvons remplacer ce dernier par un simulacre. Ce simulacre ne se connectera pas à la base de données, mais adoptera un comportement qui lui aura été dicté. Ce comportement consistera généralement à retourner certaines données en fonction de certains paramètres. Nous isolons de la sorte l'objet testé des contingences spécifiques au DAO (connexion à la base de données, etc.).

Différences entre simulacres et bouchons

L'idée d'utiliser des implémentations de test en lieu et place de véritables implémentations est une pratique assez commune. Cependant ces implémentations de tests se présentent sous

différentes formes. De façon générale, les implémentations destinées seulement aux tests sont appelées des « doublures » (en référence aux doublures dans les films d'action). Il existe différents types de doublures, notamment les simulacres et les bouchons (respectivement *mock* et *stubs*).

Les bouchons sont généralement de véritables implémentations, dans le sens où du code a été écrit pour les implémenter. Leur comportement est limité à ce qu'a prévu le développeur pour le test. Ils peuvent maintenir des informations sur les appels auxquels ils ont dû répondre (par exemple, une variable permettant de dire qu'un e-mail a été envoyé).

Les simulacres sont des objets préprogrammés avec un ensemble d'expectations qui constituent une spécification de la séquence d'appels qu'ils sont censés recevoir. Il s'agit, par exemple, de dire que la méthode `getProperty` doit être appelée en tout deux fois, la première avec le paramètre `smtp.host` et la seconde avec le paramètre `smtp.user`. Le simulacre peut aussi être programmé pour renvoyer une certaine valeur pour chacun des appels. Les comportements des simulacres pouvant être complexes, des implémentations figées (bouchons) ne peuvent convenir, d'où la nécessité d'un outillage pour générer ces objets.

L'écriture de bouchons est généralement spécifique à un projet et nécessite plus de bon sens que de technique. Notre étude se limitera donc aux simulacres.

Les simulacres d'objets avec EasyMock

Plusieurs frameworks permettent de créer facilement des simulacres au lieu de les développer manuellement. Pour les besoins de l'ouvrage, nous avons choisi EasyMock, qui est l'un des plus simples d'utilisation.

EasyMock est capable de créer des simulacres à partir d'interfaces. Une extension permet d'en fournir à partir de classes, mais nous ne l'abordons pas ici, puisque, dans le cas d'applications Spring, nous utilisons essentiellement des interfaces.

Cette section n'introduit que les fonctionnalités principales d'EasyMock. Nous utilisons la version 2.4, qui fonctionne avec Java 5. Pour plus d'informations, voir le site Web dédié au framework, à l'adresse *http://www.easymock.org*.

Les simulacres simples

Les simulacres dits simples renvoient généralement des valeurs prédéfinies. Leur comportement se rapproche de celui des bouchons. Leur avantage réside dans leur caractère dynamique, car ils sont facilement modifiables. Ils ne nécessitent pas non plus de créer de classe spécifique.

Supposons que nous désirions tester de façon unitaire la classe `ConfigurationManagerImpl` en charge de la configuration des paramètres de l'application Tudu Lists. Cette classe possède une dépendance vis-à-vis de l'interface `PropertyDAO`. Pour pouvoir l'isoler des contingences liées à l'implémentation fournie par Tudu Lists de cette interface, il est nécessaire de créer un simulacre dont nous contrôlons très exactement le comportement.

Avant de programmer notre simulacre, définissons rapidement son comportement au moyen de la méthode `getProperty`, seule utile pour notre test :

- Si le paramètre de `getProperty` vaut `key`, renvoyer un objet `Property` dont l'attribut `key` vaut `key` et dont l'attribut `value` vaut `value`.
- Dans tous les autres cas, renvoyer par défaut un objet `Property` dont les attributs `key` et `value` valent tous les deux `default`.

Maintenant que nous avons défini les comportements du simulacre, nous pouvons le définir au sein d'un cas de test :

```
package tudu.service.impl;

import static org.easymock.EasyMock.*;←1

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import tudu.domain.dao.PropertyDAO;
import tudu.domain.model.Property;

public class ConfigurationManagerImplECTest {

   private PropertyDAO propertyDAO = null;
   private ConfigurationManagerImpl configurationManager = null;

   @Before
   public void setUp() throws Exception {
      propertyDAO = createMock(PropertyDAO.class);←2
      configurationManager = new ConfigurationManagerImpl();
      configurationManager.setPropertyDAO(propertyDAO);←3
   }

   @Test
   public void getProperty() {
      Property property = new Property();
      property.setKey("key");
      property.setValue("value");
      expect(propertyDAO.getProperty("key"))
         .andStubReturn(property);←4

      Property defaultProperty = new Property();
      defaultProperty.setKey("default");
```

```
        defaultProperty.setValue("default");
        expect(propertyDAO.getProperty((String)anyObject()))
           .andStubReturn(defaultProperty);←❺

        replay(propertyDAO);←❻

        Property test = configurationManager.getProperty("key");
        assertEquals("value", test.getValue());
        test = configurationManager.getProperty("anything");
        assertEquals("default",test.getValue());

        verify(propertyDAO);←❼

    }
}
```

Pour simplifier l'écriture du code nécessaire à la définition des simulacres, des imports statiques, nouveauté introduite avec Java 5, sont utilisés (❶). Les imports statiques permettent de ne plus avoir à spécifier la classe (en l'occurrence la classe `org.easymock.EasyMock`) lors des appels à ses méthodes statiques.

Nous avons défini une méthode de gestion de contexte en utilisant l'annotation `org.junit.Before`. Par convention, nous avons appelé cette méthode `setUp`. Elle crée le simulacre et l'injecte dans le manager. La création du simulacre consiste à créer une instance d'objet implémentant l'interface `PropertyDAO` en utilisant la méthode `createMock` (❷). Cette méthode prend en paramètre l'interface que le simulacre doit implémenter, en l'occurrence `PropertyDAO`.

Une fois le simulacre créé, nous l'injectons manuellement dans le manager, puisque nous sommes dans le cadre de tests unitaires, c'est-à-dire sans conteneur (❸).

Pour que le simulacre fonctionne, il est nécessaire de spécifier le comportement de chacune de ses méthodes publiques utilisées dans le cadre du test. C'est ce que nous faisons au début de la méthode `getProperty` (❹, ❺) en utilisant la méthode statique `expect` de la classe `EasyMock`.

Au repère ❹, nous spécifions le comportement de la méthode `getProperty` de `propertyDAO` lorsque celle-ci a comme paramètre `key`, c'est-à-dire la valeur que la méthode doit renvoyer, en l'occurrence l'objet `property`. Pour cela, nous passons en paramètre à `expect` l'appel à `getProperty` proprement dit. Nous utilisons ensuite la méthode `andStubReturn` de l'objet renvoyé par `expect` pour spécifier la valeur de retour correspondant au paramètre `key`.

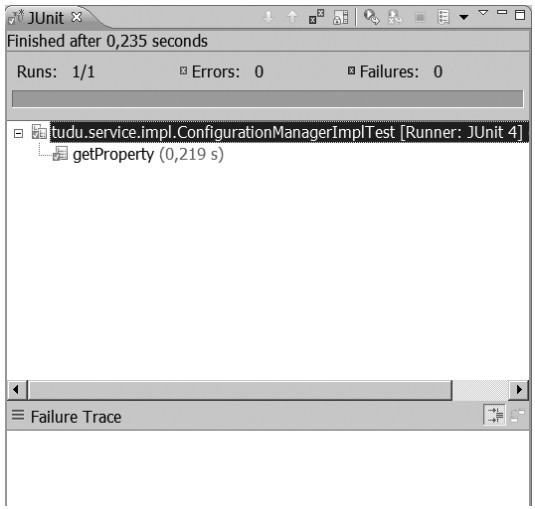
Au repère ❺, nous spécifions le comportement par défaut de `getProperty`, c'est-à-dire quel que soit son paramètre. Pour indiquer que le paramètre attendu peut être n'importe lequel, nous utilisons la méthode `anyObject` d'EasyMock. La valeur de retour est, là encore, spécifiée avec la méthode `andStubReturn`.

Pour pouvoir exécuter le test unitaire proprement dit, il suffit d'appeler la méthode `replay` en lui passant en paramètre le simulacre (❻). Nous indiquons ainsi à EasyMock que la phase d'enregistrement du comportement du simulacre est terminée et que le test commence.

Une fois le test terminé, un appel à la méthode `verify` (❼) permet de vérifier que le simulacre a été correctement utilisé (c'est-à-dire que toutes les méthodes ont bien été appelées).

Si nous exécutons notre test unitaire, nous constatons que tout se déroule sans accroc, comme l'illustre la figure 6-4.

Figure 6-4
Exécution du cas de test avec bouchon



Notons que si une méthode dont le comportement n'est pas défini est appelée, une erreur est générée. Il est possible d'affecter un comportement par défaut à chaque méthode en utilisant la méthode `createNiceMock` au lieu de `createMock`. Son intérêt est toutefois limité, car ce comportement consiste à renvoyer `0`, `false` ou `null` comme valeur de retour.

Les simulacres avec contraintes

Le framework EasyMock permet d'aller plus loin dans les tests d'une classe en spécifiant la façon dont les méthodes du simulacre doivent être utilisées.

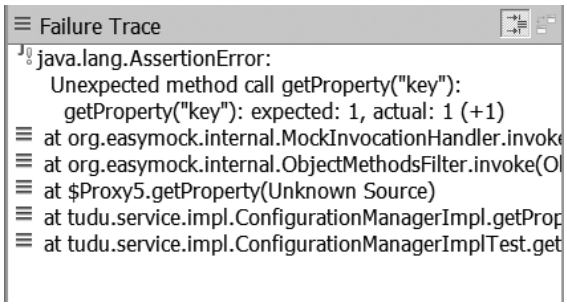
Au-delà de la simple définition de méthodes bouchons, il est possible de définir des contraintes sur la façon dont elles sont utilisées, à savoir le nombre de fois où elles sont appelées et l'ordre dans lequel elles le sont.

Définition du nombre d'appels

EasyMock permet de spécifier pour chaque méthode bouchon des attentes en termes de nombre d'appels. La manière la plus simple de définir cette contrainte consiste à ne pas définir de valeur par défaut, autrement dit à supprimer le code au niveau du repère ❺ et à remplacer la méthode `andStubReturn` par la méthode `andReturn`.

EasyMock s'attend alors à ce qu'une méthode soit appelée une seule fois pour une combinaison de paramètres donnée. Si, dans l'exemple précédent, après ces modifications, nous appelons la méthode `getProperty` avec le paramètre `key` deux fois dans notre test, une erreur d'exécution est générée, comme l'illustre la figure 6-5.

Figure 6-5
Appel inattendu à la méthode getProperty



L'erreur affichée indique que l'appel à la méthode `getProperty` est inattendu (*unexpected*). Le nombre d'appel de ce type attendu (*expected*) et effectif (*actual*) est précisé. En l'occurrence, le nombre attendu d'appel est 1, et le nombre effectif 2 (1 + 1).

De même, si la méthode `getProperty` n'est pas appelée avec le paramètre `key`, une erreur est générée puisque EasyMock s'attend à ce qu'elle soit appelée une fois de cette manière.

Pour définir le nombre de fois où une méthode doit être appelée, nous disposons des quatre méthodes suivantes, à utiliser sur le résultat produit par `andReturn` :

- `times(int nbre)` : spécifie un nombre d'appels exact.
- `times(int min,int max)` : spécifie un nombre d'appels borné.
- `atLeastOnce` : spécifie que la méthode doit être appelée au moins une fois.
- `anyTimes` : le nombre d'appels est quelconque (y compris 0).

Le code ci-dessous spécifie un nombre d'appels égal à 2 :

```
expect(propertyDAO.getProperty("key"))
    .andReturn(property).times(2);
```

Si nous exécutons notre test appelant deux fois `getProperty("key")` avec ce nouveau simulacre, nous n'obtenons plus d'erreur.

Définition de l'ordre d'appel des méthodes

L'ordre d'appel des méthodes peut être important pour tester la façon dont un objet manipule les autres. Avec EasyMock, nous pouvons définir l'ordre dans lequel les méthodes d'un simulacre doivent être appelées.

Pour tenir compte de l'ordre, il suffit de remplacer la méthode `createMock` du contrôleur par `createStrictMock`. L'ordre d'appel des méthodes est alors défini par la séquence de leur appel pour la définition du simulacre.

Pour tester l'effet de cette modification, modifions le code de notre test. Juste après le repère ❺, définissons le comportement de `getProperty` avec le paramètre `another` :

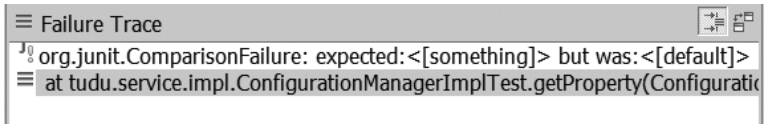
```
anotherProperty.setKey("another");
anotherProperty.setValue("something");
expect(propertyDAO.getProperty("another"))
  .andReturn(anotherProperty);
```

Introduisons ensuite le test suivant juste après le premier test de la méthode `getProperty` :

```
test = configurationManager.getProperty("another");
assertEquals("something", test.getValue());
```

Si nous effectuons cette modification sur le code précédent sans autre modification, nous obtenons le résultat illustré à la figure 6-6.

Figure 6-6
Résultat du non-respect de l'ordre d'appel



L'erreur indique que la valeur de retour attendue était celle du comportement par défaut, puisque le comportement avec le paramètre `another` a été défini après le comportement avec le paramètre `key`, qui n'a pas encore été appelé.

Les simulacres d'objets de Spring

Spring n'étant pas un framework dédié aux tests unitaires, il ne fournit pas de services similaires à ceux d'EasyMock. Cependant, afin de faciliter l'écriture de tests unitaires, Spring fournit un ensemble de simulacres prêts à l'emploi. Ces simulacres simulent des composants standards de l'API Java EE souvent utilisés dans les applications.

L'ensemble de ces simulacres est défini dans des sous-packages de `org.springframework.mock`.

Les simulacres Web

Dans le package `org.springframework.mock.web`, sont définis les simulacres nécessaires pour tester des composants Web. Nous y trouvons des simulacres pour les principales interfaces de l'API servlet, notamment les suivants :

- `HttpServletRequest`
- `HttpServletResponse`
- `HttpSession`

Dans Tudu Lists, nous disposons d'une servlet pour effectuer la sauvegarde du contenu d'une liste de todos. Le code suivant, extrait de la classe `BackupServletTest` du package `tudu.web.servlet`, montre comment les simulacres Web `MockHttpServletRequest`, `MockHttpServletResponse` et `MockHttpSession` sont utilisés pour tester cette servlet dans un cas de test JUnit classique :

```
@Test
public void doGet() throws Exception {
   Document doc = new Document();
   Element todoListElement = new Element("todolist");
```

```
    todoListElement.addContent(
      new   Element("title").addContent("Backup List"));
    doc.addContent(todoListElement);

    MockHttpServletRequest request = new MockHttpServletRequest();
    MockHttpSession session = new MockHttpSession();
    session.setAttribute("todoListDocument", doc);
    request.setSession(session);

    MockHttpServletResponse response =
      new MockHttpServletResponse();

    BackupServlet backupServlet = new BackupServlet();
    backupServlet.doGet(request, response);

    String xmlContent = response.getContentAsString();

    assertTrue(xmlContent.indexOf("<title>Backup List</title>")>0);
}
```

Le code en grisé, qui concentre l'utilisation des simulacres, montre que l'emploi de ces derniers est très aisé. Leur création ne nécessite aucun paramétrage particulier, et toutes les méthodes pour définir leur contenu puis le récupérer sont disponibles.

Autres considérations sur les simulacres

Pour terminer cette section consacrée aux simulacres d'objets, il nous semble important d'insister sur deux points fondamentaux pour bien utiliser les simulacres :

- Un cas de test portant sur un objet utilisant des simulacres ne doit pas tester ces derniers. L'objectif des simulacres est non pas d'être testés, mais d'isoler un objet des contingences extérieures afin de faciliter sa vérification.
- Un simulacre doit être conçu de manière à produire des données permettant de simuler l'ensemble du contrat de l'objet à tester. Ce contrat peut consister à retourner certaines données selon tel paramètre, mais aussi à lancer une exception selon tel autre paramètre. Le passage des tests ne démontre pas qu'un objet fonctionne correctement, mais seulement qu'il a su passer les tests.

EasyMock répond à des besoins moyennement complexes en termes de simulacres. Pour des besoins plus complexes, il est conseillé de développer spécifiquement les simulacres sans l'aide d'un framework.

En résumé

Comme pour les tests avec JUnit, les principes architecturaux de Spring facilitent la création de simulacres. D'une part, la séparation des interfaces et de leurs implémentations permet d'utiliser nativement EasyMock sans passer par une extension permettant de réaliser des simulacres à partir de classes. D'autre part, l'injection des dépendances dans le composant

pouvant être faite manuellement, donc sans le conteneur léger, il est aisé d'initialiser les collaborateurs du composant à tester, non pas avec des implémentations, mais avec des simulacres permettant de l'isoler des contingences extérieures.

Les simulacres prêts à l'emploi fournis par Spring sont très utiles en ce qu'ils simulent des classes standards de l'API Java, très utilisées dans les applications Java EE.

Les tests d'intégration

JUnit et EasyMock suffisent dans la plupart des cas à réaliser des tests unitaires. Les tests unitaires sont nécessaires mais pas suffisants pour tester le bon fonctionnement d'une application. Les tests d'intégration viennent en complément pour garantir que l'intégration entre les composants de l'application s'effectue correctement.

Dans cette section, nous allons mettre en place le test d'un DAO de Tudu Lists. Pour cela, nous utiliserons d'une part les extensions de Spring pour JUnit, afin de bénéficier de l'injection de dépendances du conteneur léger, et d'autre part le framework DbUnit pour initialiser le contenu de la base de données de tests.

Les extensions de Spring pour JUnit

Comme nous l'avons déjà indiqué, JUnit peut être utilisé pour faire des tests unitaires aussi bien que des tests d'intégration. C'est donc tout à fait naturellement que Spring fournit des extensions à JUnit afin de pouvoir tester l'intégration des composants de l'application. Ces extensions ont pour vocation d'instancier le conteneur léger afin de bénéficier de l'injection de dépendances et de la POA.

Ces extensions se présentent sous forme d'annotations, contenues dans le package `org.springframework.test.context` et ses sous-packages. Ces annotations font partie du Spring TestContext Framework, qui propose les fonctionnalités suivantes :

- Chargement d'un contexte Spring pour chaque test.
- Mise en cache automatique des contextes Spring partagés entre plusieurs tests (accélérant ainsi l'exécution globale des tests).
- Injection de dépendances dans les tests.
- Possibilité d'ajout d'objets réagissant au cycle de vie des tests.

L'utilisation d'annotations va dans le même sens que JUnit 4.x et permet aussi une meilleure réutilisabilité des composants de tests, contrairement à l'héritage de classes de test.

Spring et les versions de JUnit

Spring 2.5 supporte JUnit dans sa version 4.4 mais pas dans sa version 4.5, cela à cause de changements dans l'API interne. JUnit 4.5 est en revanche supporté par Spring 3.0. Pour le développeur de tests, il n'y a aucune différence, les deux versions de JUnit s'utilisent de la même manière.

Voici comment le test du DAO permettant de gérer les propriétés de Tudu Lists peut bénéficier du support de Spring :

```
package tudu.domain.dao.jpa;

import org.junit.runner.RunWith;
import org.springframework.test.context.ContextConfiguration;
import ➥
org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class PropertyDAOJpaTest {
   (...)
}
```

L'annotation `@RunWith` est en fait une annotation JUnit qui indique au lanceur utilisé de déléguer l'exécution du test à la classe de lanceur précisée en argument. Le `SpringJUnit4ClassRunner` implémente ainsi toute la logique d'exécution du framework de test de Spring. Utiliser ce lanceur n'ayant pas d'impact direct sur l'exécution des tests, on lance toujours ceux-ci avec les moyens habituels (lanceurs texte de JUnit, Eclipse ou outils de construction).

Le lanceur Spring va notamment extraire des informations de l'annotation `@ContextConfiguration`. Celle-ci précise qu'un contexte Spring est nécessaire au test et peut accepter des paramètres précisant quels sont les fichiers définissant ce contexte. Si aucun paramètre n'est passé à `@ContextConfiguration`, le nom du fichier de configuration sera déterminé à partir du nom de la classe de test. Dans notre cas, la classe de test s'appelle `tudu.domain.dao.jpa.PropertyDAOJpaTest` et le contexte Spring sera chargé à partir de `classpath:/tudu.domain.dao.jpa.PropertyDAOJpaTest-context.xml`.

Ce comportement par défaut est commode, mais nous préférons préciser les fichiers définissant le contexte de notre test. Ces fichiers seront au nombre de deux :

- **/tudu/conf/jpa-dao-context.xml** : déclare le contexte JPA et les DAO. Il n'est pas autonome, car il nécessite d'autres beans, notamment la connexion à la base de données (sous forme d'un `DataSource`) et l'adaptateur JPA. Ces deux beans ne sont pas directement déclarés dans ce fichier, car ils varient selon les environnements (tests d'intégration ou utilisation de l'application en production).
- **/tudu/domain/dao/jpa/test-context.xml** : complète le fichier précédent en définissant la connexion à la base de tests et l'adaptateur JPA (qui utilise Hibernate). Il définit aussi une politique de transaction car pour les tests d'intégration des DAO, en l'absence de services métier, les transactions sont ramenées sur ceux-ci.

Le fichier `jpa-dao-context.xml` est exactement le même que celui défini dans la partie sur la persistance des données, car il fait partie des fichiers de configurations fixes de Tudu Lists. Voici l'essentiel du fichier `test-context.xml` :

```
<bean id="dataSource"  class="org.springframework.jdbc.datasource.Single
ConnectionDataSource"
```

```
  >←1
  <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
  <property name="url" value="jdbc:hsqldb:mem:tudu-test" />
  <property name="username" value="sa" />
  <property name="password" value="" />
  <property name="suppressClose" value="true" />
</bean>

<bean id="jpaVendorAdapter" class="org.springframework.orm.jpa.vendor.Hibernate
JpaVendorAdapter">←2
  <property name="showSql" value="false" />
  <property name="generateDdl" value="true" />
  <property name="databasePlatform"
    value="org.hibernate.dialect.HSQLDialect" />
</bean>

<bean id="transactionManager"
    class="org.springframework.orm.jpa.JpaTransactionManager">←3
  <property
    name="entityManagerFactory"
    ref="entityManagerFactory" />
</bean>

<aop:config>
  <aop:pointcut
    id="dao"
    expression="execution(* tudu.domain.dao.jpa..*DAO*.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="dao"/>
 </aop:config>

<tx:advice id="txAdvice" transaction-manager="transactionManager">
  <tx:attributes>
    <tx:method name="*" propagation="REQUIRED" />
  </tx:attributes>
</tx:advice>
```

Le fichier commence par la définition de la connexion à la base de données sous la forme d'un `DataSource`, au repère ❶. Cette définition est particulièrement adaptée à une configuration de test : le `DataSource` ne crée qu'une connexion qui est réutilisée systématiquement, et la base de données est créée en mémoire, il n'est donc pas nécessaire de disposer d'un serveur et de le lancer.

L'adaptateur JPA est ensuite défini au repère ❷. Une implémentation Hibernate est utilisée, qui permet notamment de créer les tables à la volée. Cela se révèle particulièrement utile puisque la base de données est en mémoire et ne peut être initialisée en-dehors de l'exécution du test.

Enfin, la politique de transaction est définie au repère ❸. Il s'agit là d'une définition purement déclarative, ramenant les transactions au niveau des DAO pour le besoin des tests.

Il faut maintenant configurer le test unitaire pour qu'il utilise ces fichiers. Cela se fait en passant un tableau de chaînes de caractères au paramètre `locations` de l'annotation `@TestConfiguration` :

```
package tudu.domain.dao.jpa;

import org.junit.runner.RunWith;
import org.springframework.test.context.ContextConfiguration;
import \
org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={
    "/tudu/domain/dao/jpa/test-context.xml",
    "/tudu/conf/jpa-dao-context.xml"
})
public class PropertyDAOJpaTest {
   (...)
}
```

Le contexte de notre test va donc s'initialiser parfaitement en réutilisant un fichier de configuration de l'application et en le complétant avec un fichier dédié aux tests d'intégration des DAO.

Le test va travailler sur le DAO gérant les propriétés, `PropertyDAO`. Il faut donc que cet objet soit une propriété du test. Il suffit alors de déclarer cette propriété dans le test et de faire en sorte qu'elle soit injectée. L'injection de dépendances dans le test unitaire est bien sûr prise en charge par le lanceur Spring. La concision étant de rigueur, nous utilisons l'annotation `@Autowired` :

```
package tudu.domain.dao.jpa;

import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import \
org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import tudu.domain.dao.PropertyDAO;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={
    "/tudu/domain/dao/jpa/test-context.xml",
    "/tudu/conf/jpa-dao-context.xml"
})
public class PropertyDAOJpaTest {

   @Autowired
   private PropertyDAO propertyDAO;

}
```

Il est possible de tester rapidement l'initialisation de notre test en vérifiant que le chargement se fait correctement et en s'assurant que le DAO a été injecté. Il suffit d'ajouter la méthode de test suivante :

```
@Test
public void initOK() {
    Assert.assertNotNull(propertyDAO);
}
```

Il est clair que cette méthode ne teste pas le DAO, mais elle a le mérite de nous renseigner sur notre avancement.

Nous venons de voir comment Spring peut nous aider dans l'initialisation des tests d'intégration. C'est dans ces tests que le conteneur Spring prend toute son ampleur, car il nous permet d'élaborer très finement notre configuration de test.

L'apport de Spring ne s'arrête pas là : avec seulement quelques annotations, le contexte Spring est automatiquement chargé (et mis en cache), et le test peut se voir injecter les dépendances dont il a besoin. L'étape suivante consiste à tester proprement dit le DAO, mais cela nécessite des manipulations avec la base de données et c'est là qu'intervient DbUnit.

Utilisation de DbUnit avec Spring

DbUnit est une extension de JUnit facilitant les tests effectuant des opérations en base de données. DbUnit supporte la plupart des bases de données. Ce framework est disponible à l'adresse *http://dbunit.sourceforge.net/*.

DbUnit permet de mettre dans un état connu une base de données avant d'effectuer un test. Ce raisonnement plutôt simple est en effet le meilleur moyen d'éviter les problèmes d'initialisation et de corruption des données lors de l'exécution de tests sur une base.

DbUnit sera d'autant plus facile à intégrer dans une infrastructure de tests si une approche agile est adoptée. Voici donc un ensemble de bonnes pratiques pour les tests mettant en jeu une base de données :

- Une instance de base de données par développeur. Chaque développeur doit disposer de sa propre instance de base de données. L'idée est que chaque développeur peut effectuer ses tests sans perturber les autres. L'instance peut se trouver sur la machine du développeur. En effet, la plupart des bases gratuites sont simples à installer et peu consommatrices en ressources. La plupart des bases de données payantes proposent une édition pour le développement (Oracle propose par exemple Oracle Express Edition). Il est aussi possible de créer autant d'instances de tests qu'il y a de développeurs sur un serveur central.
- Utiliser des jeux de données minimalistes. Les données doivent seulement permettre de tester toutes les éventualités, pas de mesurer la rapidité d'exécution. Il est donc préférable de remplir les tables avec au maximum quelques dizaines d'enregistrements, et ce afin que les tests s'exécutent le plus rapidement possible. L'optimisation des requêtes (jointures, index) doit plutôt être faite avec de véritables données, bien que la question des performances doive toujours être à l'esprit des développeurs !

- Tous les tests doivent être indépendants. Il ne faut pas qu'un test dépende des données d'un autre test. Chaque test doit donc avoir son propre jeu de données, et aucune autre phase d'initialisation des données ne doit être nécessaire.

Au cœur de DbUnit se trouve la notion de jeu de données (*dataset*, qui correspond à l'interface `org.dbunit.dataset.IDataSet`). DbUnit synchronise la base de données à partir d'implémentations de `IDataSet`. La manière la plus commune de créer un jeu de données est d'écrire un fichier XML et de laisser DbUnit créer l'objet correspondant.

Prenons, par exemple, la table contenant les propriétés de l'application Tudu Lists. Les annotations JPA posées sur l'entité nous indiquent que la table s'appelle `property` et que la propriété `key` correspond au champ `pkey`. La propriété `value` est automatiquement mise en correspondance avec la colonne de même nom. Voici le fichier XML correspondant :

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>

  <property pkey="smtp.host" value="some.url.com" />

</dataset>
```

Ce format est appelé « plat » (*flat*) dans la nomenclature des `IDataSet` de DbUnit. On remarque qu'au nom de la table correspond le nom de la balise et qu'aux propriétés correspondent les attributs de la balise. Chaque balise `property` correspond à un enregistrement dans la table correspondante.

Si ce fichier s'appelle `property.xml`, le code suivant permet de charger ce fichier sous forme de `IDataSet` :

```
IDataSet dataSet = new FlatXmlDataSet(new File("property.xml"));
```

À partir d'un objet `IDataSet`, il est possible de compter le nombre d'enregistrements de chacune des tables, de récupérer les valeurs pour chaque colonne de chaque enregistrement de chaque table, etc. L'opération qio nous intéresse consiste à mettre le contenu du jeu de données dans notre base. L'insertion doit passer par une connexion JDBC ou un `DataSource`. La plupart de nos composants utilisant un `DataSource`, nous optons pour cette solution.

Voici un exemple de code pour effectuer l'insertion de notre jeu de données :

```
import javax.sql.DataSource;←1
import org.dbunit.database.DatabaseDataSourceConnection;
import org.dbunit.dataset.IDataSet;
import org.dbunit.dataset.xml.FlatXmlDataSet;
import org.dbunit.operation.DatabaseOperation;

(...)

   IDataSet dataSet = new FlatXmlDataSet(
      new File("property.xml")
   );←2
   DataSource dataSource;←3
```

```
    // récupération du DataSource
    (...)
    DatabaseDataSourceConnection datasourceConnection =
       new DatabaseDataSourceConnection(
          dataSource
       );←4
    DatabaseOperation.CLEAN_INSERT.execute(
       datasourceConnection,
       dataSet
    );←5
```

Il faut dans un premier temps effectuer un ensemble d'imports de packages, principalement de DbUnit (1).

Le repère 2 reprend la création du `IDataSet`. Nous initialisons ensuite le `DataSource` (3). Cette partie n'est pas détaillée pour l'instant (Spring nous permettra de combler cette lacune par la suite). Il faut ensuite créer une connexion telle que l'attend DbUnit, et ce à partir du `DataSource`. C'est ce qui est effectué au repère 4, en utilisant `DatabaseDataSourceConnection`, qui fait partie de l'API DbUnit.

Le repère 5 constitue la partie la plus intéressante de notre exemple. C'est là que s'effectue l'injection du jeu de données dans la base de données. Pour cela, un appel statique sur `DatabaseOperation` nous permet d'effectuer une insertion « propre » (*clean insert*). Dans notre cas, cela consistera à vider la table `property` et à la remplir avec les enregistrements décrits dans le fichier XML. L'opération de *clean insert* est l'opération d'injection de données par défaut effectuée par DbUnit : elle vide toute table impliquée dans le jeu de données.

DbUnit propose une API très riche et des fonctionnalités plus intéressantes les unes que les autres, mais nous avons vu là l'essentiel qui nous intéresse. L'idée est maintenant d'effectuer cette injection de données juste avant nos méthodes de test. Le code précédent a donc sa place dans une méthode annotée avec `@Before` afin d'être exécutée avant chaque méthode de test. Reprenons notre test unitaire où nous l'avons laissé :

```
package tudu.domain.dao.jpa;

(...)

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={
    "/tudu/domain/dao/jpa/test-context.xml",
    "/tudu/conf/jpa-dao-context.xml"
})
public class PropertyDAOJpaTest {

   @Autowired
   private PropertyDAO propertyDAO;

   @Autowired
   private DataSource dataSource
```

```
    @Before
    public void setUp() {
       IDataSet dataSet = new FlatXmlDataSet(
          new File("property.xml")
       );
       DatabaseOperation.CLEAN_INSERT.execute(
          new DatabaseDataSourceConnection(dataSource),
          dataSet
       );
    }

    @Test
    public void getPropertyOK() {
       Property prop = propertyDAO.getProperty("smtp.host");
       Assert.assertNotNull(prop);
       Assert.assertEquals("smtp.host",prop.getKey());
       Assert.assertEquals("some.url.com",prop.getValue());
    }

    (...)

}
```

Cette méthode d'initialisation s'intègre parfaitement avec le fonctionnement de nos tests d'intégration, rendant l'utilisation de DbUnit très facile. L'exemple montre aussi le test de la méthode `getProperty` pour une clé existant en base. Malgré son aspect plutôt simple, rendu possible notamment par le support JUnit de Spring, ce test nous assure que le fichier de configuration des DAO de Tudu Lists est correct et que la méthode `getProperty` fonctionne correctement.

Il est aussi possible de tester un autre contrat de notre méthode, celui qui consiste à dire qu'elle renvoie un objet nul si la clé n'existe pas en base :

```
@Test
public void getPropertyNOK() {
    Property prop = propertyDAO.getProperty("some.dummy.key");
    Assert.assertNull(prop);
}
```

Ce test paraît anodin, mais renvoyer une exception serait un comportement tout aussi possible. Un quelconque changement de comportement sera donc immédiatement détecté grâce à ce test.

Deux autres méthodes sont à tester dans le DAO : `saveProperty` et `updateProperty`. DbUnit est encore d'une aide précieuse pour s'assurer du bon déroulement des méthodes. Voici le test pour la méthode `saveProperty` :

```
@Test
public void saveProperty() throws Exception {
```

```
    Property toSave = new Property();
    toSave.setKey("some.key");
    toSave.setValue("some.value");←1
    propertyDAO.saveProperty(toSave);←2
    DatabaseDataSourceConnection dsConnection =
       new DatabaseDataSourceConnection(
          dataSource
       );←3
    IDataSet databaseDataSet = dsConnection.createDataSet();←4
    ITable tablePersonne = databaseDataSet.getTable("property");←5
    assertEquals(2,tablePersonne.getRowCount());←6
}
```

Le test commence par la création de l'objet propriété à persister (1). Au repère 2, la méthode à tester est appelée. Il faut ensuite, comme pour l'injection du jeu de données, travailler avec l'API de DbUnit. Cela passe par la création de la connexion DbUnit, *via* le `DataSource` (injecté dans le test par Spring), au repère 3. L'objet de connexion permet de créer un `IDataSet` à partir du contenu de la base de données (4). À partir de ce `IDataSet`, il est possible de récupérer la table qui nous intéresse (5) et de connaître le nombre d'enregistrements dans cette table afin d'effectuer une vérification (6). Notre jeu de données XML injectant une seule ligne dans la table `property`, il faut qu'il y ait deux enregistrements après l'appel à la méthode `saveProperty`. DbUnit pourrait nous permettre de pousser plus loin nos vérifications en allant jusqu'à vérifier les valeurs des colonnes pour chacun des enregistrements. Nous n'irons pas jusque-là, la vérification du nombre d'enregistrements dans la table étant suffisante.

L'ensemble de ces vérifications permet de tester efficacement notre DAO ; cependant, les appels à l'API DbUnit sont un peu fastidieux. Tudu Lists contient donc une classe utilitaire permettant de faire rapidement les opérations les plus courantes (injection d'un jeu de données, récupération du nombre d'enregistrements dans une table, etc.). Cette classe s'appelle `DbUnitHelper` ; elle nécessite seulement un `DataSource` comme dépendance. Un Bean `dbUnitHelper` est déclaré dans le contexte Spring. Nous ne détaillerons pas l'implémentation de cette classe, qui encapsule des appels basiques mais quelque peu verbeux à l'API DbUnit. Cette classe peut aussi être utilisée dans la méthode `setUp` afin d'effectuer l'injection du jeu de données.

Voici maintenant l'apparence de notre test unitaire :

```
(...)
public class PropertyDAOJpaTest {

   @Autowired
   private PropertyDAO propertyDAO;

   @Autowired
   private DbUnitHelper helper;

   @Before
   public void setUp() throws Exception {
```

```
        IDataSet dataSet = new FlatXmlDataSet("property.xml");
        helper.doCleanInsert(dataSet);
    }

    @Test
    public void saveProperty() throws Exception {
        Property toSave = new Property();
        toSave.setKey("some.key");
        toSave.setValue("some.value");
        propertyDAO.saveProperty(toSave);
        assertEquals(2,helper.getTable("property").getRowCount());
    }
}
```

L'utilisation du `DbUnitHelper` rend le test unitaire un peu moins verbeux et moins tributaire de DbUnit.

Le test de la méthode `updateProperty` est lui aussi très simple :

```
@Test
public void updateProperty() throws Exception {
    Property toUpdate = new Property();
    toUpdate.setKey("smtp.host");
    toUpdate.setValue("some.other.host");
    propertyDAO.updateProperty(toUpdate);
    ITable table = helper.getTable("property");
    Assert.assertEquals(
        "some.other.host",
        table.getValue(0, "value").toString()
    );
}
```

En résumé

Les tests d'intégration utilisant JUnit sont facilités pour les applications qui utilisent Spring. Le Spring TestContext Framework permet d'ajouter du comportement aux classes de tests unitaires. Il gère alors le chargement du contexte Spring, sa mise en cache s'il est partagé par plusieurs classes de tests et l'injection de dépendances dans le test unitaire.

Toutes ces fonctionnalités permettent de recréer un contexte d'exécution pour les tests d'intégration. Une division judicieuse des fichiers Spring permet de réutiliser ceux-ci pour différents environnements. Les tests des DAO de Tudu Lists travaillent ainsi sur une base de données en mémoire, sans que le fichier de définition des DAO n'ait eu à subir de modifications.

La puissance du conteneur léger Spring permet d'intégrer élégamment DbUnit. Il est alors très aisé de mettre la base de données dans un état adapté pour tous les tests et donc de tester de façon fiable tout composant interagissant avec la base de données.

Réagir à l'exécution des tests

Le Spring TestContext Framework propose un système d'écouteur à l'exécution des classes de tests. C'est par ce biais que l'injection de dépendances dans un test est possible. Ce mécanisme très puissant permet d'ajouter facilement du comportement « autour » de l'exécution des tests. L'impact sur la classe de test unitaire est minime, et seul l'ajout d'une annotation paramétrée correctement est nécessaire (pas d'héritage ou d'interface à implémenter).

Voici la définition de l'interface d'écouteur :

```
package org.springframework.test.context;

public interface TestExecutionListener {

   void prepareTestInstance(TestContext testCtx) throws Exception;

   void beforeTestMethod(TestContext testCtx) throws Exception;

   void afterTestMethod(TestContext testCtx) throws Exception;

}
```

Les méthodes sont exécutées suite à la création de l'objet de test puis avant et après chaque méthode de test.

Chaque test dispose de sa propre pile d'écouteurs. Il est possible de la paramétrer avec l'annotation `org.springframework.test.context.TestExecutionListeners` à laquelle on passe un tableau de `TestExecutionListeners` :

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@TestExecutionListeners({MyTestExecutionListener.class})
public class MyTest {
   (...)
}
```

L'annotation `@TestExecutionListeners` ne peut évidemment fonctionner que si les annotations `@RunWith` et `@ContextConfiguration` sont positionnées. Si l'annotation `@TestExecutionListeners` n'est pas positionnée, l'écouteur d'injection de dépendances (fourni par Spring) est positionné par défaut. Cette annotation n'est donc à utiliser que dans le cas où l'on souhaite positionner ces propres écouteurs.

Ce principe d'écouteur peut sembler bien complexe et surtout redondant compte tenu de l'existence du système d'annotations JUnit réagissant à l'exécution du test (`@Before` et `@After`). Son utilité est toutefois justifiée par sa plus grande réutilisabilité. Un écouteur peut encapsuler un code complexe qui alourdirait un test si ce code apparaissait directement dans une méthode annotée avec `@Before` ou `@After`. Une fois l'écouteur codé, il peut être ajouté simplement *via* l'annotation `@TestExecutionListeners`, ce qui est très peu intrusif.

Nous allons illustrer l'écriture d'un écouteur par l'amélioration de notre infrastructure de test d'intégration. Nous avons précédemment effectué l'injection d'un jeu de données en base de données dans une méthode annotée avec `@Before`. Cela convenait parfaitement au test de notre DAO. Mais imaginons que nous voulions tester d'autres composants interagissant avec la base de données. Le test de ces composants nécessiterait aussi d'avoir une base de données dans un état connu, et cela passerait bien évidemment par DbUnit.

Ce besoin de réutilisabilité nous conduit à penser que l'écriture d'un `TestExecutionListener` serait tout à fait adaptée. Cet écouteur serait positionné sur des tests qui précisent un jeu de données à injecter. Nous pouvons, par exemple, définir une interface que ces tests devraient implémenter pour localiser le fichier XML de ce jeu de données :

```
public interface DataSetLocator {

   public String getDataSet();

}
```

L'interface `DataSetLocator` définit (sous la forme d'une chaîne de caractères) la localisation du `IDataSet`. Nous aurions pu définir une annotation ou adopter un fonctionnement par défaut (localiser le fichier XML en fonction du nom de la classe de test), mais cette approche par interface nous semble un bon compromis.

Nous pouvons définir l'écouteur effectuant l'injection de la manière suivante :

```
(...)
public class CleanInsertTestExecutionListener
  implements TestExecutionListener {

  public void beforeTestMethod(TestContext ctx) throws Exception {
    if(ctx.getTestInstance() instanceof DataSetLocator) {←1
      DataSetLocator test = (DataSetLocator) ctx.getTestInstance();
      ApplicationContext appCtx = ctx.getApplicationContext();
      DbUnitHelper helper = null;
      if(appCtx.containsBean("dbUnitHelper")) {
        helper = (DbUnitHelper) appCtx.getBean("dbUnitHelper");
      } else {
        DataSource ds = (DataSource) appCtx.getBean("dataSource");
        helper = new DbUnitHelper(ds);
      }←2
      IDataSet dataSet = helper.getDataSet(test.getDataSet());←3
      helper.doCleanInsert(dataSet);←4
    }
  }

  public void prepareTestInstance(TestContext ctx) throws Exception
  {
    // aucune opération←5
  }
```

```
    public void afterTestMethod(TestContext ctx) throws Exception {
      // aucune opération←6
    }

}
```

L'injection du jeu de données se fait avant chaque méthode de test : c'est pourquoi la méthode `beforeTestMethod` est implémentée. Celle-ci vérifie que le test unitaire implémente bien l'interface `DataSetLocator` afin de pouvoir localiser le jeu de données (❶). Le code précédant le repère ❷ permet de disposer d'un `DbUnitHelper`, soit en le récupérant dans le contexte Spring, soit en en créant un en utilisant le `DataSource`. Le jeu de données est ensuite créé à partir de l'information donnée par le test (❸), puis injecté dans la base de données (❹). Le `DbUnitHelper` nous assiste bien dans toutes ces tâches.

Les deux autres méthodes de l'interface `TestExecutionListener` n'ayant pas d'utilité dans notre cas, elles ont une implémentation vide (❺, ❻).

Voici maintenant le test unitaire du DAO propriété :

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={
   "/tudu/domain/dao/jpa/test-context.xml",
   "/tudu/conf/jpa-dao-context.xml"
})
@TestExecutionListeners{
   CleanInsertTestExecutionListener.class,←1
   DependencyInjectionTestExecutionListener.class←2
})
public class PropertyDAOJpaTest implements DataSetLocator {

   (...)

   public String getDataSet() {
      return "/tudu/domain/dao/dataset/property.xml";←3
   }

}
```

La méthode `setUp`, effectuant l'injection du jeu de données, n'est plus nécessaire. Le test unitaire définit dans sa pile d'écouteurs non seulement l'écouteur effectuant l'injection de données (❶), mais aussi celui faisant l'injection de dépendances (❷). Celui-ci est positionné par défaut en l'absence de l'annotation `@TestExecutionListeners`, mais il n'est pas ajouté automatiquement si l'on précise sa propre pile.

Enfin, le test unitaire implémentant l'interface `DataSetLocator` définit la méthode `getDataSet`, qui doit retourner la localisation du jeu de données au format XML plat de DbUnit (❸).

Cette plongée dans les mécanismes du Spring TestContext Framework nous a permis d'écrire un composant facilement réutilisable pour l'injection de données en base de données. Cette approche est particulièrement élégante pour intégrer des composants tels que DbUnit et ajouter du comportement de façon complètement transversale à des tests unitaires.

Conclusion

Ce chapitre a montré comment tester une application Spring avec JUnit. Du point de vue des tests unitaires, le code fondé sur Spring ne nécessite pas d'extension particulière à JUnit, hormis l'utilisation le cas échéant d'un framework spécifique, comme EasyMock, pour simuler les objets dont dépend le composant à tester.

Spring fournit par ailleurs des simulacres prêts à l'emploi pour certains composants techniques de l'API Java EE. Pour les tests unitaires, seul le code applicatif est utilisé, sans l'aide du conteneur léger de Spring. L'injection des dépendances est effectuée manuellement dans les tests.

Pour les tests d'intégration, JUnit nécessite plusieurs extensions, car, dans ce cas, le conteneur léger doit être utilisé. À cette fin, Spring fournit le TestContext Framework qui se charge de la gestion du contexte des tests. Un utilitaire tel que DbUnit peut alors facilement être intégré pour tester, par exemple, les composants interagissant avec la base de données.