Un livre de Wikilivres.

Introduction au test logiciel

Introduction

Avant d'entrer dans le vif du sujet, revenons à la notion de qualité logicielle pour détourer quels aspects de celle-ci nous allons traiter. Voyons ensuite les *symptômes* du développeur qui ne teste pas ou teste mal. Enfin, nous expliquerons pourquoi ce livre devrait susciter l'intérêt du lecteur concerné.

La qualité logicielle ?

Une notion subjective

Selon les points de vue, la qualité logicielle n'a pas la même définition.

- Pour le développeur, il s'agit d'abord de voir la qualité du code-source : celui-ci est-il modulaire, générique, compréhensible, documenté ?
- Pour le responsable du système d'information un logiciel doit plutôt être facile à administrer et ne pas corrompre la sécurité globale du SI.
- L'utilisateur, lui, s'attend plutôt à ce qu'un logiciel remplisse d'abord son besoin sans être trop coûteux, en étant si possible ergonomique voire esthétique.
- Pour le financier, il s'agit plutôt de savoir si le logiciel se vend bien, s'il permet de faire du profit ou s'il permet d'améliorer la compétitivité d'une structure.

Autant de préoccupations différentes pour un même produit. Dans ce qui suit, nous allons plutôt nous

intéresser à la qualité du logiciel telle qu'elle est perçue par le développeur.

La nécessité d'assurer un suivi de la qualité logicielle

Pour vous convaincre de la nécessité d'intégrer dans toute démarche de développement (qu'elle se fasse dans le cadre d'une entreprise, d'un projet institutionnel, d'une réalisation bénévole en logiciel libre) une approche raisonnable de la qualité logicielle, nous pourrions lister les plus fameuses défaillances de l'histoire de l'informatique et d'autres domaines de l'ingénierie.

Sans nous lancer dans le récit exhaustif de ces grandes catastrophes techniques dont les plus fameuses ont coûté des vies humaines tandis que d'autres ont gaspillé des milliards de budget, citons tout de même quelques cas où une défaillance informatique s'est avérée responsable au moins en partie d'une catastrophe[1] :

- la sonde Mariner 1 dont l'explosion a été déclenché 294,5 secondes après son décollage en raison d'une défaillance des commandes de guidage ;
- le satellite Mars Climate Orbiter a raté sa mise en orbite autour de la planète Mars pour s'écraser sur la planète rouge : certains paramètres avaient été calculés en unités de mesure anglo-saxonnes et transmises à l'équipe de navigation qui attendait ces données en unités du système métrique. Coût total du projet : 327,6 million de $;
- Therac-25 est une machine de radiothérapie qui coûta la vie à plusieurs patients ;
- Le projet Virtual Case File est un fiasco : le développement, commandé par le FBI, fut entièrement abandonné au bout de 5 ans et 100 millions de $;
- Le vol 501 de la fusée Ariane 5 : elle explosa lors de son premier vol, 40 secondes après son décollage, emportant avec elle les 370 millions de dollars investis dans le projet et les satellites qu'elle devait transporter.

Ces projets sont bien sûr hors du commun et sont bien différents de ceux que chaque entreprise, ingénieur ou développeur rencontrera dans sa carrière. Souvent, le risque causé par une panne est plus limité. Toutefois, de nombreux projets nécessitent un haut niveau de sureté de fonctionnement :

- Contrôle de centrale nucléaire,
- Émetteur de télécommunication/télévision (niveau d'émission des ondes),
- En général, tout logiciel qui peut contrôler un équipement électrique/électronique utilisant une puissance électrique potentiellement dangereuse,
- De façon générale, tous les logiciels contrôlant des équipements impliquant des êtres vivants (médical, transport, ...).

Néanmoins, même pour les projets ne requérant aucun niveau de sécurité, la présence de bogues provoquant la défaillance d'un système peut engendrer des coût imprévus voire mener à l'échec d'un projet, ce qui est déplaisant pour tous les intervenants. En 2002, une étude commandée par le National Institute of Standards and Technology conclue que les bogues et les erreurs logiciels coûtent chaque année 59 billions de $ à l'économie américaine soit 0,6 % du PIB.

Les mauvaises pratiques ou « À qui s'adresse ce livre ? »

Si, lorsque que vous constatez l'apparition d'un bogue dans un programme que vous êtes en train de développer, vous adoptez le comportement suivant :

1. Placer des appels d'affichage[2] un peu partout.
2. Lancer le programme pour évaluer l'affichage produit en espérant trouver des informations permettant d'isoler la zone d'où le bogue provient.

3. Corriger le bogue en modifiant et relançant le programme autant de fois que nécessaire.
4. Effacer tous les appels d'affichage qui polluent le code pour revenir à un code propre.
5. Recommencer à chaque fois qu'un bogue apparaît ou réapparaît.

Cette démarche, en plus d'être coûteuse en temps, est inefficace. Elle est difficile à mettre en œuvre dans une grande base de code lorsqu'on ne sait pas d'où le bogue peut venir. De plus, tout ce travail est perdu étant donné que si un bogue similaire réapparaît, il faudra remettre les traces dans le code en espérant ne pas avoir déjà oublié comment le bogue était apparu, et comment il avait été corrigé. Si c'est ainsi que vous procédez, ce livre est fait pour vous !

Si lorsque votre code atteint une complexité suffisamment grande, vous commencez à craindre de faire des modifications sur des portions de code que vous avez écrite il y a longtemps parce que vous ne savez pas si vos modifications risquent de casser quelque-chose sans que vous vous en rendiez compte ; si vous en avez marre de passer 20 % de votre temps à développer et 80 % de temps à chasser les bogues ; si vous perdez de l'argent parce que pour un produit donné, vous passez plus de temps à corriger des bogues gratuitement lors de la période de garantie que de temps à développer le produit en étant payé, alors ce livre est pour vous.

Pour un développeur, pratiquer des tests et vérifications permet de relire le code source, repérer les erreurs et apprendre à les éviter.

Pourquoi vérifier et tester ?

La vérification et les tests sont deux disciplines de la qualité logicielle. Ces techniques, si elles sont adoptées et appliquées rigoureusement et avec intelligence dès le début d'un projet peuvent améliorer significativement la qualité logicielle.

L'adoption d'une pratique de test systématique apportent de multiples avantages :

- Un développeur peut travailler plus sereinement. Si une modification introduit un bogue dans le code, le développeur en sera tout de suite informé. Les erreurs sont corrigées en amont : on évite ainsi les surprises ;
- En évitant ainsi les surprises, on a une meilleure visibilité sur le temps à consacrer pour le reste du développement. Ainsi, le gestionnaire ne verra pas ses ressources réquisitionnées par la découverte, le lendemain de la mise en production, de nombreux bogues qui auraient pu être corrigés plus tôt.
- Le produit final est de meilleure qualité, cela est apprécié par l'utilisateur final. On réduit ainsi le coût de la garantie.

Ce livre s'adresse donc aux développeurs qui ignorent les aspects techniques d'une démarche de test logiciel intégrée au processus de développement. C'est une initiation à ces pratiques qui est proposée tout au long de cet ouvrage.

Après la lecture de ce livre, vous serez en mesure :

- de choisir, en fonction des technologies que vous utilisez, les outils qui vous permettront de tester et de vérifier le code que vous écrivez ;
- d'écrire vos propres tests pour vérifier la qualité de votre code ;
- de concevoir vos applications de manière à les rendre testables ;
- de vérifier la qualité de vos tests ;
- d'intégrer ces éléments dans votre démarche globale de développement.

Remarquons que ces deux disciplines peuvent ne pas être suffisantes et être complétées par d'autres solutions adéquates visant à assurer la qualité logicielle. Remarquons enfin que si la qualité d'un logiciel au sens où nous la traitons ici peut jouer un rôle, elle n'est en rien garante du succès commercial d'un projet.

Quels sont les pré-requis ?

La lecture de ce livre nécessite d'avoir tout de même une expérience même courte du développement en utilisant la programmation orientée objet. Il est bon d'avoir une expérience sur un projet suffisamment complexe pour avoir perçu ce qui est évoqué plus haut.

Références

1. Les plus curieux d'entre vous pourront trouver d'autres cas dans l'article « 20 famous software disasters » publié sur devtopics.com (http://www.devtopics.com/20-famous-software-disasters/) [archive] pour ce qui concerne l'informatique. Pour l'ingénierie en général, voir la catégorie catégorie « Engineering failures ».
2. System.out.println(), echo, printf etc.

Tests unitaires

Dans ce chapitre, nous allons voir ce qu'est un « test unitaire ». Il s'agit du test le plus couramment utilisé et le plus important. Nous verrons comment créer des tests unitaires pour votre code et comment les utiliser. Nous verrons enfin les limitations de ces tests.

Introduction aux tests unitaires

L'objectif d'un test unitaire est de permettre au développeur de s'assurer qu'une unité de code ne comporte pas d'erreur de programmation. C'est un *test*, donc les vérifications sont faites en exécutant une petite partie (une « *unité* ») de code. En programmation orientée objet, l'unité est la classe[1]. Un test est donc un programme qui exécute le code d'une classe pour s'assurer que celle-ci est correcte.

Concrètement, un test, c'est du code. À chaque classe d'une application, on associe une autre classe qui la teste.

Les frameworks de type xUnit

Pour écrire des tests unitaires, vous avez à votre disposition des frameworks qui vont vous faciliter l'écriture des tests. Vous n'aurez plus qu'à écrire les classes de tests et c'est le framework qui se chargera de les trouver, de les lancer et de vous donner les résultats ou les erreurs qui ont été détectées.

En Java
: JUnit (http://junit.org/) [archive] est le framework de type xUnit le plus utilisé pour Java. Il est tellement utilisé qu'il est livré avec la plupart des IDE. C'est ce framework sur lequel nous allons nous appuyer pour l'exemple ci-après.

En C++
: Cutter (http://cutter.sourceforge.net/) [archive], Google propose Google C++ Testing Framework (http://code.google.com/p/googletest/) [archive], la fameuse bibliothèque Boost comprend la Boost Test Library (http://www.boost.org/doc/libs/release/libs/test/index.html) [archive].

En Python
: La distribution de base de Python intègre unittest (http://docs.python.org/library/unittest.html) [archive] mais il existe aussi PyUnit (http://pyunit.sourceforge.net/) [archive].

En PHP

Les développeurs PHP utilisent PHPUnit (http://www.phpunit.de/) [archive] ou SimpleTest (http://simpletest.org/) [archive]. SimpleTest est utilisé dans Drupal (http://drupal.org/simpletest) [archive] depuis sa version 7. Un nouveau framework de test existe pour Php 5.3+ : Atoum (https://github.com/atoum/) [archive].

En Ruby

Ruby intègre Test::Unit (http://www.ruby-doc.org/stdlib/libdoc/test/unit/rdoc/classes/Test/Unit.html) [archive].

En D

Le langage intègre nativement le mot-clé unittest.

En Groovy

voir Unit Testing (http://groovy.codehaus.org/Unit+Testing) [archive]

En JavaScript

le framework jQuery utilise qunit (https://github.com/jquery/qunit) [archive], Jarvis (http://jarvis.tmont.com/) [archive], jfUnit (http://felipenmoura.org/projetos/jfunit/home.php) [archive], google-js-test (http://code.google.com/p/google-js-test/) [archive]

Dans d'autres langages

Il existe des frameworks équivalents dans la plupart des langages : vous pouvez les découvrir dans l'article « List of unit testing frameworks » sur Wikipedia anglophone.

Introduction à JUnit

Ici, nous allons présenter succinctement JUnit afin d'illustrer l'utilisation d'un framework de type xUnit. Même si on ne pratique pas Java au quotidien, cela permet de voir les principes généraux qu'on retrouve dans toutes les implémentations de ce framework. Nous n'allons pas nous étendre plus longtemps sur la question étant donné que de nombreux tutoriels sont disponibles sur la Toile pour les différents langages.

JUnit 4 requiert Java 1.5 car il utilise les annotations. JUnit 4 intègre JUnit 3 et peut donc lancer des tests écrits en JUnit 3.

Écrire un test

Avec JUnit, on va créer une nouvelle classe pour chaque classe testée. On crée autant de méthodes que de tests indépendants : imaginez que les tests peuvent être passés dans n'importe quel ordre (i.e. les méthodes peuvent être appelées dans un ordre différent de celui dans lequel elles apparaissent dans le code source). Il n'y a pas de limite au nombre de tests que vous pouvez écrire. Néanmoins, on essaye généralement d'écrire au moins un test par méthode de la classe testée.

Pour désigner une méthode comme un test, il suffit de poser l'annotation `@Test`.

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

public class StringTest {

    @Test
    public void testConcatenation() {
        String foo = "abc";
        String bar = "def";
        assertEquals("abcdef", foo + bar);
    }

    @Test
    public void testStartsWith() {
```

```
        String foo = "abc";
        assertTrue(foo.startsWith("ab"));
    }

}
```

Les assertions

Via `import static org.junit.Assert.*;`, vous devez faire appel dans les tests aux méthodes statiques assertTrue, assertFalse, assertEquals, assertNull, fail, etc. en fournissant un message :

```
?
```

Votre environnement de développement devrait vous permettre de découvrir leurs signatures grâce à l'auto-complétion. À défaut, vous pouvez toutes les retrouver dans la documentation de l'API JUnit (http://www.junit.org/apidocs/org/junit/Assert.html) [archive].

Attention à ne pas confondre les assertions JUnit avec « `assert` ». Ce dernier est un élément de base du langage Java[2]. Par défaut, **les assertions Java sont ignorées** par la JVM à moins de préciser `-ea` au lancement de la JVM.

Lancer les tests

Pour lancer les tests, vous avez plusieurs possibilités selon vos préférences :

- La plus courante : lancer les tests depuis votre IDE
- Utiliser l'outil graphique
- Lancer les tests en ligne de commande
- Utiliser un système de construction logiciel (comme Ant ou Maven pour Java)

Factoriser les éléments communs entre tests

On peut déjà chercher à factoriser les éléments communs à tous les tests d'une seule classe. Un test commence toujours par l'initialisation de quelques instances de formes différentes pour pouvoir tester les différents cas. C'est souvent un élément redondant des tests d'une classe, aussi, on peut factoriser tout le code d'initialisation commun à tous les tests dans une méthode spéciale, qui sera appelée avant chaque test pour préparer les données.

Si on considère l'exemple ci-dessus, cela donne :

```
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.*; //or import org.junit.Before; + import org.junit.After;

public class StringTest {

    private String foo;
    private String bar;

    @Before // avec cette annotation, cette méthode sera appelée avant chaque test
    public void setup() {
        foo = "abc";
        bar = "def";
```

```
    }

    @After
    public void tearDown() {
        // dans cet exemple, il n'y a rien à faire mais on peut,
        // dans d'autres cas, avoir besoin de fermer une connexion
        // à une base de données ou de fermer des fichiers
    }

    @Test
    public void testConcatenation() {
        assertEquals("abcdef", foo + bar);
    }

    @Test
    public void testStartsWith() {
        assertTrue(foo.startsWith("ab"));
    }

}
```

Remarquons que la méthode annotée @Before, est exécutée avant *chaque* test, ainsi, chaque test est exécuté avec des données saines : celles-ci n'ont pu être modifiées par les autres tests.

On peut également souhaiter factoriser des éléments communs à plusieurs classes de tests différentes, par exemple, écrire un test pour une interface et tester les différentes implémentations de ces interfaces. Pour cela, on peut utiliser l'héritage en écrivant les tests dans une classe de test abstraite ayant un attribut du type de l'interface et en écrivant ensuite une classe fille par implémentation à tester, chacune de ces classes filles ayant une méthode @Before différente pour initialiser l'attribut de la classe mère de façon différente.

Vérifier la levée d'une exception

Il peut être intéressant de vérifier que votre code lève bien une exception lorsque c'est pertinent. Vous pouvez préciser l'exception attendue dans l'annotation @Test. Le test passe si une exception de ce type a été levée avant la fin de la méthode. Le test est un échec si l'exception n'a pas été levée. Vous devez donc écrire du code qui *doit* lever une exception et pas qui *peut* lever une exception.

```
@Test(expected = NullPointerException.class)
public void methodCallToNullObject() {
    Object o = null;
    o.toString();
}
```

Désactiver un test temporairement

Au cours du processus de développement, vous risquez d'avoir besoin de désactiver temporairement un test. Pour cela, plutôt que de mettre le test en commentaire ou de supprimer l'annotation `@Test`, JUnit propose l'annotation `@Ignore`. En l'utilisant, vous serez informé qu'un test a été ignoré, vous pouvez en préciser la raison.

```
@Ignore("ce test n'est pas encore prêt")
@Test
public void test() {
    // du code inachevé
}
```

Écrire de bon tests

Idéalement, les tests unitaires testent une classe et une seule et sont indépendants les uns des autres. En effet, si une classe A est mal codée et qu'elle échoue au test unitaire de A, alors B qui dépend de A (parce qu'elle manipule des instances de A ou hérite de A) va probablement échouer à son test alors qu'elle est bien codée. Grâce à cette propriété on assure, que si un test échoue, c'est bien la classe testée qui est fautive et pas une autre. Nous verrons qu'assurer cette propriété peut être complexe.

Selon les langages, on a souvent des conventions pour nommer les classes de tests. En Java, on choisit souvent d'appeler *MaClassTest* le test de la classe *MaClass*. Les tests peuvent être commentés si le déroulement d'un test est complexe.

Évidemment, un bon test assure la qualité de l'intégralité d'une classe, et pas seulement d'une partie de celle-ci. Nous verrons comment vérifier cela dans la partie « qualité des tests ».

Limites du test unitaire

Les classes abstraites

Comme l'écriture d'un test unitaire requiert d'instancier une classe pour la tester, les classe abstraites (qui sont par définition non-instantiables) ne peuvent être testées. Remarquons que cela n'empêche pas de tester les classes filles qui héritent de cette classe et qui sont concrètes.

Une solution consiste à créer, dans la classe de test, une classe interne héritant de la classe abstraite la plus simple possible mais qu'on définit comme concrète. Même si cette classe n'est pas représentative des classes filles qui seront finalement écrites, cela permet tout de même de tester du code de la classe abstraite.

Les attributs invisibles

Si un attribut est privé et qu'il ne propose pas d'accesseur public, il n'est pas possible de voir son état sans moyen détourné. Une façon consiste ici à créer une classe interne au test qui hérite de la classe à tester et y ajoute les accesseurs nécessaires. Il suffit alors de tester cette nouvelle classe.

Les méthodes privées

Comme vous ne pouvez pas faire appel à une méthode privée, vous ne pouvez vérifier son comportement facilement. Comme vous connaissez l'implémentation de la classe que vous testez (boîte blanche). Vous pouvez ruser et essayer de trouver une méthode publique qui fait appel à la méthode privée.

Une autre méthode, plus brutale et rarement utilisée, consiste à utiliser l'API de réflexion du langage (java.reflect pour Java) pour charger la classe, parcourir ses méthodes et toutes les passer en *public*.

Les classes ayant des dépendances

Une classe A qui dépend d'une autre classe B pour diverses raisons :

- A hérite de B
- A a un attribut de type B
- une méthode de A attend un paramètre de type B

Du point de vue UML, un trait relie A et B. Or, peut-être que B n'a pas encore été implémentée ou qu'elle n'est pas testée. En plus, il faut veiller à conserver l'indépendance des tests entre eux, et donc, que notre test

de A ne dépende pas de la réussite des tests de B.

Pour cela, nous pouvons utiliser des doublures de test : elles permettent, le temps du test de A, de satisfaire les dépendances de A sans faire appel à B. De plus, l'utilisation d'une doublure permet de simuler un comportement de B vis-à-vis de A, qu'il serait peut-être très difficile à simuler avec la véritable classe B (par exemple, des cas d'erreurs rares comme une erreur de lecture de fichier).

À défaut, il faudrait pouvoir indiquer dans les tests qu'on écrit que la réussite du test de A dépend du test de B, parce que si les deux échouent : il ne faudrait pas chercher des erreurs dans A alors qu'elles se trouvent dans B. Ainsi, le framework, prenant en compte les dépendances, pourrait remonter à l'origine du problème sans tromper le développeur.

Des outils complémentaires

Nous avons vu quelques limitations que nous pouvons compenser par des astuces de programmation ou l'utilisation de doublures de tests que nous verrons prochainement dans ce livre. Toutefois, il existe des frameworks de tests qui essaient de combler les lacunes de xUnit. Ils méritent qu'on y jette un coup d'œil.

Citons l'exemple de TestNG (http://testng.org/) [archive], pour Java qui permet notamment d'exprimer des dépendances entre tests ou de former des groupes de tests afin de ne pouvoir relancer qu'une partie d'entre eux.

Le logiciel BlueJ représente les classes de tests sous la forme de rectangles verts, exécutables via un clic droit. Elles sont enregistrables à partir d'opérations effectuées à la souris, et héritent toujours de :

```
public class ClasseTest extends junit.framework.TestCase {
```

Ses méthodes de test doivent avoir un nom commençant par "test".

Références

1. Dans d'autres paradigmes, c'est plutôt la fonction ou la procédure qui peuvent être considérés comme l'unité testée
2. voir http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html

Doublures de test

Nous avons vu que les tests unitaires pouvaient rapidement devenir difficile à écrire pour diverses raisons. Nous allons voir que dans de nombreux cas, nous pouvons résoudre ces problèmes par l'utilisation de doublures. Il s'agit d'objets qui seront substitués aux objets attendus pour obtenir un comportement qui rend les tests plus facile à écrire.

Pourquoi utiliser une doublure

Parmi les problèmes qu'on peut rencontrer lors de l'écriture de tests, nous pouvons énumérer :

1. **La classe testée fait appel à un composant difficile ou coûteux à mettre en place**. Typiquement : une base de données. Si on veut tester cette classe, il faut, au début du test, créer une base de données,

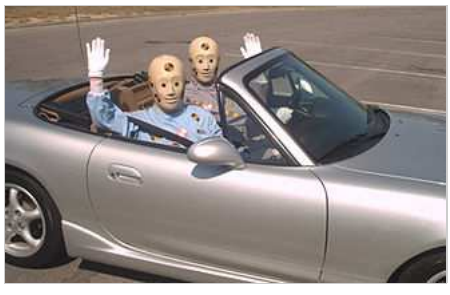
insérer des données pertinentes (ce qui peut être très compliqué si la base contrôle l'intégrité référentielle et que le schéma est complexe ou si des déclencheurs sont appelés), et, à la fin du test, les supprimer. On risque en plus, si le test est interrompu (erreur, levée d'exceptions) de ne pas supprimer les données et d'avoir une base dans un état inconnu.

2. **Le test vise à vérifier le comportement d'une classe dans une situation exceptionnelle**. Par exemple, on veut vérifier le comportement d'un code réseau lorsqu'une déconnexion survient. Il est évidemment absurde d'imaginer un développeur qui attend le bon moment de son test pour débrancher le câble réseau de sa machine... Typiquement, il s'agit de vérifier la réaction d'un code lors de la levée d'une exception.
3. **La classe testée fait appel à un composant qui n'est pas encore disponible de façon stable**. C'est souvent le cas lorsqu'une application est découpée en couches et que les différents développeurs développent ces couches en parallèle. Il se peut alors qu'un composant ne soit tout simplement pas disponible ou que son interface soit soumise à des changements très fréquents. Imaginons une application Web simple, 3-tiers. On a un premier composant, l'interface utilisateur qui fait appel à un second composant, le code métier qui fait appel à un troisième composant chargé de la persistance (via une base de données). Comment tester le code métier quand le composant persistance est toujours en cours de développement ? Comment tester l'interface graphique si le code métier n'est toujours pas stable ?
4. **Le test fait appel à du code lent**. Il est souhaitable d'avoir un jeu de test permettant de s'assurer de la qualité globale du système qui ne prenne pas des heures. Si la classe testée fait appel à autre classe lente (parce que les calculs sont très complexes, parce qu'il y des accès disques ou tout autre raison). Comment tester cette classe seulement sans être ralenti par la lenteur de la seconde ?
5. **La classe testée fait appel à du code non-déterministe**. Typiquement, la classe se comporte différemment selon l'heure courante, ou des nombres générés aléatoirement. On imagine pas non plus un développeur changer l'horloge système de son système au cours de ses tests ou attendre la nuit pour lancer ces tests afin de vérifier si la classe se comporte bien pendant la nuit.
6. **Il semble nécessaire d'ajouter du code à une classe pour mener des tests**. Une application est, en général, déjà suffisamment compliquée sans qu'on ait besoin de rajouter du code dont certains se demanderont à quoi il sert puisqu'il n'est jamais appelé dans le reste de l'application. Il faut éviter que du code spécifique aux tests parasite le code d'une classe applicative.

Dans chacun de ces cas, nous allons voir que l'utilisation de doublures peut résoudre le problème posé.

Présentation des doublures

S'il fallait faire une analogie pour expliquer les doublures, on pourrait parler des mannequins utilisés lors des essais de choc automobile. Ils jouent le rôle de doublures d'êtres humains pour *vérifier le comportement* de la voiture dans une *situation exceptionnelle : l'accident* (on retrouve le cas n°2). Les doublures reproduisent la structure et la forme des originaux (les mannequins ont le poids, la morphologie de véritables êtres humains pour que les tests soient fiables) mais la voiture ne s'en rend pas compte, elle réagit comme si elle transportait des humains.



Des mannequins (« dummy » en anglais) sont utilisés dans les essais de choc automobile pour vérifier que les passagers sont bien protégés en cas d'accident

Quand on parle de test logiciel, une doublure est un objet qui va, au moment de l'exécution, remplacer un objet manipulé par du code. Le code manipulateur ne s'en rend pas compte, une bonne doublure trompe le code appelant qui doit agir comme s'il manipulait un objet normal. Écrire des doublures ne requiert aucune compétence technique particulière autre qu'écrire du code dans le langage de l'application testée ou d'apprendre à utiliser

de petites bibliothèques. En revanche, la création de doublures efficaces fait appel à la *ruse* du développeur. Il faut maîtriser le langage et savoir jouer avec le polymorphisme et les petites subtilités de chaque langage pour réaliser les doublures les plus simples possibles.

Comme nous allons le voir, il existe plusieurs types de doublures.

Les bouchons (ou « stubs »)

Un bouchon est le type le plus simple de doublure, il s'agit d'une classe, écrite à la main. Son implémentation tient compte du contexte exact pour lequel on a besoin d'une doublure. Quand on écrit un bouchon on considère tout le système comme une boîte blanche, on sait comment il est implémenté on en tient compte pour écrire le bouchon le plus simple possible, avec le minimum de code nécessaire pour qu'il puisse remplir son rôle de doublure.

Un exemple

Dans l'exemple suivant, nous allons tomber dans le cas n°3. Dans cet exemple, nous sommes chargés de développer la partie messagerie utilisateur d'un logiciel, c'est un autre développeur qui se charge de la partie authentification utilisateur du système. Son travail est de fournir au logiciel une implémentation de l'interface `IdentificationUtilisateur` que vous devrez utiliser dans votre messagerie.

```
/** permet de vérifier qu'une personne à bien accès au système */
public interface IdentificationUtilisateur {

    /** vérifie qu'un utilisateur existe et que son mot de passe est le bon
     * @param identifiant l'identifiant de la personne à authentifier
     * @param motDePasse le mot de passe de la personne
     * @return vrai si l'utilisateur existe et que le mot de passe est le bon
     */
    boolean identifier(String identifiant, String motDePasse);
}
```

Vous avez fait votre travail et avez développé une classe MessagerieUtilisateur qui gère la messagerie, elle utilise `IdentificationUtilisateur` pour vérifier les identifiants et mot de passe de l'utilisateur.

```
public class MessagerieUtilisateur {

    protected IdentificationUtilisateur identification;

    public MessagerieUtilisateur(IdentificationUtilisateur identification) {
        this.identification = identification;
    }

    public List<Message> lireMessages(String identifiant,
                                      String motDePasse)
                                      throws IdentificationException {

        boolean identification = identification.identifier(identifiant, motDePasse);
        if (identification) {
            // code qui retourne la liste des messages pour cet utilisateur
        } else {
            throw new IdentificationException();
        }
    }
}
```

Soucieux de la qualité de votre implémentation, vous vous lancez dans la création d'un test unitaire sous JUnit. Il devrait, au moins en partie, ressembler à la classe suivante :

```
1 /** Test de la classe MessagerieUtilisateur */
2 public class MessagerieUtilisateurTest {
3
4     @Test
5     public void testLireMessages() {
6
7         // problème : comment instancier messagerie, il nous manque un paramètre
8         MessagerieUtilisateur messagerie = new MessagerieUtilisateur(???);
9
10        try {
11            messagerie.lireMessages("toto", "mdp");
12        } catch (IdentificationException e) {
13            fail();
14        }
15
16        try {
17            messagerie.lireMessages("toto", "mauvais_mot_de_passe");
18            // une exception aurait due être levée
19            fail();
20        } catch (IdentificationException e) {
21            assertTrue(true);
22        }
23    }
24 }
```

Nous voici devant le problème, le développeur chargé d'implémenter l'interface `IdentificationUtilisateur` dont vous avez besoin ne peut pas encore vous la fournir, il a d'autres tâches prioritaires. Vous allez devoir vous en passer. C'est à ce moment que vous allez créer un bouchon. Cette classe va se substituer à celle qui manque le temps du test.

```
/** doublure de la classe IdentificationUtilisateur
 *  c'est un bouchon qui est utilisé dans MessagerieUtilisateurTest
 */
public class IdentificationUtilisateurStub implements IdentificationUtilisateur {

    /** seul le compte "toto" avec le mot de passe "mdp" peut s'identifier */
    @Override
    boolean identifier(String identifiant, String motDePasse) {
        if ("toto".equals(identifiant) && "mdp".equals(motDePasse) {
            return true;
        }
        return false;
    }
}
```

Il ne reste plus qu'à écrire dans le test ligne 7

```
MessagerieUtilisateur messagerie = new MessagerieUtilisateur(new IdentificationUtilisateu
```

Vous pouvez maintenant dérouler le test, sans plus attendre. Le bouchon a permis de résoudre le problème de dépendance non-satisfaite envers un composant qui n'est pas disponible.

Les simulacres (ou « mock objects »)

Contrairement aux bouchons, les simulacres ne sont pas écrits par le développeur mais générés par l'usage d'un outil dédié. Cet outil permet de générer un simulacre à partir de la classe originale. Le simulacre généré est d'un type similaire à la classe originale et peut donc la remplacer. On peut ensuite, par programmation, définir comment la doublure doit réagir quand elle est manipulée par la classe testée.

Java
: Il en existe plusieurs mais le plus utilisé est Mockito (http://mockito.org/) [archive]. D'autres solutions sont plus abouties comme Mockachino (http://code.google.com/p/mockachino/) [archive] ou JMockit (http://code.google.com/p/jmockit/) [archive] ;

C++
: Google propose le Google C++ Mocking Framework (http://code.google.com/p/googlemock/) [archive] ;

Groovy
: Gmock (http://gmock.org/) [archive] ;

Ruby
: mocha (http://mocha.rubyforge.org/) [archive]

JavaScript
: SinonJS (http://sinonjs.org/) [archive]

Dans les autres langages
: vous pouvez lire l'article « List of mock object frameworks » de Wikipedia anglophone.

Introduction à Mockito

Mockito est une bibliothèque Java qui permet de créer des simulacres.

Reprenons l'exemple précédent. Cette fois, grâce à mockito, nous n'aurons plus besoin d'écrire le bouchon que nous avons utilisé précédemment. Nous n'allons pas utiliser un bouchon mais un simulacre, créé juste au moment du test.

Dépendance Maven (?)

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-
core</artifactId>
  <scope>test</scope>
</dependency>
```

sur central (http://repo1.maven.org/maven2/org.mockito/mockito-core/) [archive]

Dépendance Maven (?)

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-
all</artifactId>
  <scope>test</scope>
</dependency>
```

sur central (http://repo1.maven.org/maven2/org.mockito/mockito-all/) [archive]

```
import static org.mockito.Mockito.*; // permet d'utilise

/** Test de la classe MessagerieUtilisateur avec un simu
public class MessagerieUtilisateurTest {
    @Test
    public void testLireMessages() {

        // mockIdentification sera notre simulacre, comm
        // c'est bien un objet qui implémente l'interfac
        IdentificationUtilisateur mockIdentification;

        // la méthode statique mock de la class Mockito
        // le simulacre.
        mockIdentification = mock(IdentificationUtilisat

        // maintenant, il faut spécifier le comportement
        // pour que le test se déroule bien
        when(mockIdentification.lireMessages("toto", "mo
        when(mockIdentification.lireMessages("toto", "ma


        // notre simulacre est prêt à être utilisé
        MessagerieUtilisateur messagerie = new Messageri

        try {
```

```
            messagerie.lireMessages("toto", "mdp");
        } catch (IdentificationException e) {
            fail();
        }

        try {
            messagerie.lireMessages("toto", "mauvais_mot_de_passe");
            // une exception aurait due être levée
            fail();
        } catch (IdentificationException e) {
            assertTrue(true);
        }
    }
}
```

L'outil de génération de simulacres nous a permis d'éviter d'écrire une nouvelle classe (le bouchon). Cela nous a permis d'écrire le test plus vite sans ajouter de code excessif.

Limites de Mockito

Mockito, en interne, joue avec la réflexion et l'héritage de Java. Par conséquent, on ne peut pas *mocker*

- les méthodes equals() et hashCode(), Mockito les utilise en interne et les redéfinir pourrait dérouter Mockito ;
- les méthodes privées (`private`) ;
- les classes déclarées `final` ;
- les méthodes déclarées final (Mockito exécutera le code original de la classe sans tenir compte de vos instructions sans vous en informer donc soyez vigilant) ;
- les méthodes de classes (`static`).

Les trois dernières impossibilités sont dues à la façon dont Java charge les classes en mémoire. Il est possible de remédier à ces limitations justement en changeant la façon dont Java charge les classes. C'est ce que propose PowerMock (http://powermock.org) [archive] qui étend Mockito.

Les espions (ou « spy »)

Un espion est une doublure qui enregistre les traitements qui lui sont fait. Les tests se déroulent alors ainsi :

1. On introduit une doublure dans la classe testée
2. On lance le traitement testé
3. Une fois finie, on inspecte la doublure pour savoir ce qu'il s'est passé pour elle. On peut vérifier qu'une méthode a bien été appelée, avec quels paramètres et combien de fois etc.

L'intérêt de cette méthode est manifeste lorsqu'il s'agit de tester des méthodes privées. On ne peut vérifier la valeur retournée ou que les paramètres passés successivement au travers des différentes méthodes sont bons. Dans ce cas, on laisse l'espion passer successivement pour toutes les méthodes privées et c'est seulement à la fin que l'espion nous rapporte ce qu'il s'est vraiment passé.

Une classe espionne peut être codée à la main mais la plupart des outils de création de simulacres évoqués plus haut créent des doublures qui sont aussi des espions.

Un exemple, avec Mockito

Les doublures créées par Mockito sont aussi des espions. Réécrivons notre test en utilisant cette possibilité.

```
/** Test de la classe MessagerieUtilisateur avec un espion */
public class MessagerieUtilisateurTest {

    public void testLireMessages() throws Exception {

        // mockIdentification sera notre simulacre, comme on peut le voir
        // c'est bien un objet qui implémente l'interface IdentificationUtilisateur
        IdentificationUtilisateur mockIdentification;

        // la méthode statique mock de la class Mockito permet de créer
        // le simulacre.
        mockIdentification = Mockito.mock(IdentificationUtilisateur.class);

        // maintenant, il faut spécifier le comportement attendu par notre simulacre
        // pour que le test se déroule bien
        Mockito.when(mockIdentification.identifier("toto", "mdp")).thenReturn(true);
        Mockito.when(mockIdentification.identifier("toto", "mauvais_mot_de_passe")).thenR


        // notre simulacre est prêt à être utilisé
        // étape 1 : on introduit la doublure
        MessagerieUtilisateur messagerie = new MessagerieUtilisateur(mockIdentification);

        // étape 2 : on lance le traitement
        messagerie.lireMessages("toto", "mdp");
        messagerie.lireMessages("toto", "mauvais_mot_de_passe");

        // étape 3 : Le code a été exécuté, voyons ce que rapporte l'espion
        // vérifions que la méthode identifier() a bien été appelée exactement une fois a
        Mockito.verify(mockIdentification, times(1)).identifier("toto", "mdp");
        // et exactement une fois avec ces paramètres
        Mockito.verify(mockIdentification, times(1)).identifier("toto", "mauvais_mot_de_pa
    }
}
```

Mockito propose bien sûr d'autres utilisation de `verify()`, permet de vérifier l'ordre dans lequel les méthodes ont été appelées etc. Pour voir toutes les possibilités, il faut se référer à la documentation de Mockito (http://mockito.googlecode.com/svn/tags/latest/javadoc/org/mockito/Mockito.html) [archive].

Les substituts (ou « fake »)

Un substitut est une doublure écrite à la main qui implémente le comportement attendu d'une classe mais de façon plus simple. Contrairement au bouchon qui est écrit spécifiquement pour un test, le substitut a vocation à être suffisamment générique pour être utilisé dans plusieurs tests. Il est donc plus compliqué à écrire que le bouchon mais à l'avantage d'être réutilisable, le temps supplémentaire passé à écrire cette doublure peut être rentabilisé si elle est réutilisée dans d'autres tests.

Un exemple

Revenons à notre premier exemple, celui du bouchon. Notre test est très court, supposons que nous avons d'autres tests à mener sur notre classe `MessagerieUtilisateur`. Par exemple, on voudrait tester le comportement de la classe avec plusieurs utilisateurs différents, et vérifier que les messages sont bien stockés. Nous avons la possibilités de créer d'autres bouchons pour chacun de ces tests, mais essayons plutôt d'écrire un substitut qui pourra servir dans chaque test.

```
/** doublure de la classe IdentificationUtilisateur
 *  c'est un substitut qui est utilisé dans MessagerieUtilisateurTest
```

```
 */
public class IdentificationUtilisateurStub {

    /** les comptes de tous les utilisateurs, identifiants et mot de passes associés */
    Map<String, String> comptes = new HashMap<String, String>();

    /** vrai si le compte a été ajouté et le mot de passe est valide */
    @Override
    boolean identifier(String identifiant, String motDePasse) {
        boolean result = false;
        if (comptes.containsKey(identifiant)) {
            if (comptes.get(identifiant).equals(motDePasse)) {
                result = true;
            }
        }
        return result;
    }

    /** ajoute un compte à l'ensemble des comptes valides */
    public void ajouterCompte(String identifiant, String motDePasse) {
        comptes.put(identifiant, motDePasse);
    }
}
```

Ce substitut devrait nous permettre d'écrire les différents tests évoqués. Il suffira, au début du test, d'ajouter zéro, un ou plusieurs comptes selon le besoin.

Les fantômes (ou « dummy »)

Il s'agit du type de doublure le plus simple, il s'agit simplement d'un objet qui ne fait rien puisqu'on sait que, de toute façon, dans la méthode testée, il n'est pas utilisé. Il ya plusieurs façon simples de créer un fantôme :

- Il peut s'agir d'un objet qui implémente une interface et laisse toutes les implémentations vides[1] ;
- Un simulacre généré ;
- Un objet quelconque, par exemple une instance de Object en Java (on peut très bien faire `Object fantome = new Object();`)
- Parfois, l'absence de valeur convient très bien (`null`).

Comment utiliser les doublures

Finalement, reprenons la totalité des cas évoqués et voyons comment les doublures peuvent nous aider.

1. **La classe testée fait appel à un composant difficile ou coûteux à mettre en place**. Il faut une doublure pour ce composant. Ce composant doit avoir une façade, en général, pour tester une partie. Si on reprend l'exemple de la base de données, un bouchon, qui renvoie des données minimum écrites en dur juste pour le test (comme dans l'exemple de bouchon) devrait faire l'affaire. Comme c'est en général un gros composant, la façade peut être très grande (beaucoup de méthodes à implémenter) mais il y a de fortes chances que le test ne fasse vraiment appel qu'à une petite partie de celles-ci. Ainsi, il ne faut pas hésiter à créer un semi-fantôme, c'est à dire une doublure qui fait bouchon pour les quelques méthodes utilisées et qui en fait ne fait rien pour tout le reste. S'il y a beaucoup de tests, et que le composant n'est pas une façade trop développée, peut-être qu'un substitut du composant évitera la création de beaucoup de bouchons.
2. **Le test vise à vérifier le comportement d'une classe dans une situation exceptionnelle**. On peut très bien remplacer l'implémentation d'une classe par une doublure qui remplace l'implémentation d'une méthode par une levée d'exception.

3. **La classe testée fait appel à un composant qui n'est pas encore disponible de façon stable**. Utiliser une doublure à la place du composant le temps d'écrire les tests et de les passer, remplacer la doublure par le composant final quand celui-ci est prêt (et garder la doublure dans un coin, ça peut toujours servir).
4. **Le test fait appel à du code lent**. S'il s'agit d'une partie isolée, on peut en faire un substitut plus rapide. S'il s'agit d'une application réseau (beaucoup d'entrées/sorties), il faut plutôt envisager de faire un substitut bas niveau seulement pour la couche réseau, et de passer les messages d'objets à objets (censés être distants mais en fait tout deux en mémoire dans le même processus) directement sans passer par le réseau. S'il s'agit d'algorithmes complexes, qui prennent beaucoup de temps, utiliser une doublure qui renvoie des données codées en dur, voire utiliser un algorithme plus efficace pour faire une estimation et renvoyer des données du même ordre de grandeur que celles attendues.
5. **La classe testée fait appel à du code non-déterministe**. Faire une doublure qui renvoie des données de façon déterministe. Pour l'horloge, diriger vers plusieurs doublures qui renvoient chacune une heure donnée, codée en dur. Pour le générateur de nombres aléatoires, une doublure qui renvoie les éléments connus d'une liste, voire toujours le même nombre. Une autre façon est d'utiliser un générateur qui permet de fixer une base de départ (une *graine*). Lorsque deux générateurs aléatoires sont créé avec la même graine, ils génèrent les mêmes données aléatoires. Ce comportement déterministe permet de prédire le comportement des classes testées et d'écrire les tests adéquats.
6. **Il semble nécessaire d'ajouter du code à une classe pour mener des tests**. A priori, ce code qui doit être ajouté permet en général de faire des vérifications sur ce qui a été produit. Plutôt que de modifier cette classe, il doit pouvoir être possible de générer un simulacre et de l'utiliser comme espion. Sinon, créer une nouvelle sous-classe dans les tests pour ajouter les comportements manquants.

Quoiqu'il en soit, il faut toujours essayer d'utiliser les doublures avec parcimonie. Elles donnent l'illusion de tester l'application alors qu'en fait, on teste des coquilles vides qui ne permettent finalement pas de présager du comportement final de l'application, quand celle-ci utilisera intégralement les implémentations en lieu et place des doublures. Abandonner l'idée d'utiliser une doublure si son code devient trop complexe par rapport à la complexité du test qu'elle est censée permettre.

Tester une classe abstraite

L'objectif est de tester les méthodes concrètes d'une classe non-instantiables. Supposons l'existence d'une classe A, abstraite, et de B et C des implémentations concrètes. Plusieurs possibilités :

- écrire un test abstrait pour A dont les tests de B et C hériteront pour tester également les méthodes héritées. Dans ce cas, le même test (si les méthodes concrètes de A ne sont pas redéfinies) de chaque méthode de A sera rejoué pour chaque sous-classe de A, ce qui est une perte de temps ;
- écrire un test concret de A. Dans ce cas, il faut dans ce test créer une implémentation de A ;
  - créer un bouchon pour A : dans ce cas, à chaque ajout d'une méthode abstraite dans A, il faudra compléter la doublure qui peut avoir de nombreuses implémentations vides, ce qui est déplaisant ;
  - créer un simulacre de A et simplement lui dire de renvoyer les vrais résultats lors de l'appel.

Mockito permet cela grâce à son support de mock *réels* partiels (http://docs.mockito.googlecode.com/hg/org/mockito/Mockito.html#16) [archive] (c'est à dire des simulacres qui peuvent adopter le vrai comportement de la classe simulée) :

```
@Test
public testUneMethodeConcrete() {
    // préparation des données en entrées pour le test
    int a = 1;

    A mock = Mockito.mock(A.class);
```

```
        Mockito.when(mock.uneMethodeConcrete(a)).thenCallRealMethod();

        // assertions....
        assertEquals(2, mock.uneMethodeConcrete(a));
    }
```

Implications de l'utilisation de doublures

- Bénéfices apportés par les doublures
  - Découplage entre les classes
  - Reproductibilité accrue
  - Augmentation significative de la qualité des tests étant donné la possibilité de tester les situations exceptionnelles
- Inconvénients des doublures
  - Temps de développement initial
  - Temps de maintenance : nécessité pour les doublures de suivre les changements dans les classes originales
  - Lorsqu'une classe fonctionne avec une doublure, cela ne garantit pas que la classe fonctionnera quand il s'agira d'interagir avec la véritable implémentation. Pour vérifier cette seconde propriété, il faudra recourir à un test d'intégration.

Limites des doublures

Dans certains codes, il sera difficile de substituer un objet par sa doublure. C'est particulièrement le cas si, par exemple, on souhaite remplacer un objet qui est un attribut privé de la classe, qu'il est affecté dès l'initialisation de la classe par la classe elle-même et qu'on a pas de possibilité de le modifier depuis l'extérieur.

```
/* comment remplacer attributPrive par sa doublure pour tester
 * methode() ? Dans ce cas, ce n'est pas possible.
 */
public class ClasseTestee {

    private final UneClasseQuelconque attributPrive = new UneClasseQuelconque();

    public methode() {
        // code qui fait appel a attributPrive
    }
}
```

La manière dont cette classe est écrite limite l'utilisation de doublures, elle réduit la testabilité du code et montre que les possibilités d'utilisation de doublures peuvent être réduites voire nulles. Nous verrons dans le chapitre « Tests et conception » comment éviter ce genre d'écueil.

Références

1. Voir aussi le patron de conception Null Object

Tests et conception

Nous avons vu qu'il pouvait parfois être difficile de tester du code, comme dans la section « Limites des doublures » si la structure de celui-ci ne s'y prêtait pas. Dans le présent chapitre, nous allons voir les bonnes et les mauvaises pratiques qu'il faut connaître lorsqu'on écrit du code afin de rendre celui-ci *testable*. On utilise parfois le terme « testabilité ».

Faciliter l'instanciation

Nous l'avons vu au premier chapitre sur les tests unitaires : tester, c'est exécuter le code et donc instancier. Aussi si une classe est difficile à instancier, cela complique le test. Si le constructeur de la classe a beaucoup de paramètres, il faudra créer autant d'objet et eux-même peuvent nécessiter encore d'autres instances. Tout cela peut aboutir à un cauchemar d'instanciation.

S'il est si difficile d'instancier une classe, il y a peut-être un problème de conception. Est-ce que la Loi de Déméter est respectée ? Est ce qu'il n'y a pas un problème de séparation des préoccupations ? Si cette classe a besoin d'autant de dépendances, ne fait-elle pas trop de choses ? Ne devrait-elle pas déléguer une partie du travail ?

Certaines questions peuvent être résolues en respectant certains patrons de conceptions, notamment les patrons GRASP définissant la répartitions des responsabilités des classes d'un logiciel (voir en particulier **le faible couplage** et **la forte cohésion**).

Permettre l'utilisation de doublures

Design par interfaces

Décrivez les principaux composants par leurs contrats, c'est à dire par des interfaces. Il est plus facile de créer un bouchon ou un substitut quand c'est une interface qui est attendue plutôt qu'une classe.

Sous-classes

Avec la programmation orientée objet il ne faut pas hésiter à subdiviser la hiérarchie d'une classe afin de concevoir des classes simples à utiliser et donc à tester.

Quand une interface possède plusieurs classes d'implémentation, il peut être utile de créer une classe parente commune à toutes ces classes afin d'y implémenter les comportements communs et faciliter également l'ajout éventuel d'une nouvelle classe d'implémentation. Cela éviter de répéter du code plusieurs fois et permet de tester une fois pour toutes les méthodes communes à ces classes.

Permettre l'injection de dépendances

Nous avons vu que certaines classes peuvent rendre difficile, voire impossible, l'utilisation de doublures.

```
/* comment remplacer attributPrive par sa doublure pour tester
 * methode() ? Dans ce cas, ce n'est pas possible.
 */
public class ClasseTestee {

    private final UneClasseQuelconque attributPrive;

    public ClasseTestee() {
        attributPrive = new UneClasseQuelconque();
    }
```

```
    public methode() {
        // code qui fait appel a attributPrive
    }
}
```

Le problème vient du fait que la classe qui détient le comportement est également responsable de l'instanciation. Ici, il y a deux solutions pour permettre l'injection de dépendance. La première est de ne pas instancier dans le constructeur mais de demander à l'appelant de passer l'objet en argument du constructeur. Toutefois, si ce constructeur est appelé à différents endroits dans le reste de l'application, il faudra dupliquer le code d'instanciation (on perd en factorisation). La deuxième solution est de garder ce constructeur et d'en créer un autre, qui, lui, permet d'injecter la dépendance. Ainsi, l'application utilisera le premier tandis que dans les tests, ou pourra utiliser le deuxième pour injecter la doublure.

```
public class ClasseTestee {

    private final UneClasseQuelconque attributPrive;

    public ClasseTestee(UneClasseQuelconque uneClasseQuelconque) {
        attributPrive = uneClasseQuelconque;
    }

    public ClasseTestee() {
        this(new UneClasseQuelconque());
    }

    public methode() {
        // code qui fait appel a attributPrive
    }
}
```

Permettre l'isolation

Éviter les états globaux

Les états globaux sont des données partagées de façon transverses dans toute l'application. Typiquement, il s'agit d'instances de type singletons ou variables globales.

Permettre l'observation

- Permettre la redéfinition par héritage
- Laisser la structure interne visible (pas de champs private, mais plutôt protected). Indiquer ce qui a été rendu visible uniquement à des fins de testabilité. Par exemple, les ingénieurs de Google ont choisi d'ajouter l'annotation `@VisibleForTesting` dans l'API de Guava (http://guava-libraries.googlecode.com/svn/trunk/javadoc/com/google/common/annotations/VisibleForTesting.html) [archive].
- Regardez ce que retourne vos méthodes. Les objets retournés permettent-ils de vérifier que tout s'est bien passé ?

Qualité des tests

Classification des tests

Un des concepts fondamentaux du développement logiciel est la séparation des préoccupations. C'est un principe à respecter également lorsqu'on écrit des tests : un test doit répondre à une préoccupation précise. Aussi, il est impératif lors de l'écriture d'un test de savoir :

- quel est le périmètre du (sous-)système testé,
- à quel niveau d'abstraction se place-t-on. Si l'application est divisée en couches (architecture n-tiers), avec l'API de quelle couche le test est écrit,
- quelles sont les suppositions (on ne re-teste pas toute l'application à chaque fois). Si on teste une couche donnée, la plupart du temps, on suppose que les couches inférieures fonctionnent (elle-mêmes seront testées dans d'autres tests).

Toutes ces informations doivent être précisées dans la documentation du test ou de la classe de test. Pour vous aider à faire le tri, la classification ci-dessous définit quelques critères qui peuvent aider à caractériser un test.

Tests « boîtes blanches » et tests « boîtes noires »

On peut parler de tests « boîtes blanches » ou de tests « boîtes noires » selon la visibilité que le développeur qui écrit les tests a sur le code testé.

Test boîte blanche
: Le testeur, pour écrire son test, tient compte de la structure interne du code testé et de l'implémentation. Avec la connaissance de l'implémentation, le testeur peut facilement visualiser les différentes branches et les cas qui provoquent des erreurs. Éventuellement, le test peut faire appel à des méthodes protégées voire privées.

Test boîte noire
: Le testeur ne tient pas compte de l'implémentation mais ne s'appuie que sur la partie publique de l'élément testé pour écrire les tests.

Avec les tests boîte blanche, il est plus facile d'atteindre une bonne couverture étant donné que l'accès au code testé permet de voir les différentes branches et les cas d'erreurs. L'avantage des tests en boite noire est, qu'une fois écrits, l'implémentation de la classe testée peut changer sans nécessiter avoir à mettre à jour le test. Ainsi, en Java, un test boîte noire écrit en faisant appel aux méthodes d'une interface peut permettre de tester toutes les classes qui réalisent cette interface.

Tests fonctionnels et tests non-fonctionnels

On parle de tests fonctionnels quand il s'agit de vérifier qu'une classe permet bien de remplir avec succès l'objectif fixé par un cas d'utilisation [1] donné. Un test fonctionnel permet de répondre à la question « est-ce que le code permet de faire ça ? » ou « est-ce que cette fonctionnalité attendue est bien fonctionnelle ? ».

Par opposition, les tests non-fonctionnels vérifient des propriétés qui ne sont pas directement liées à une utilisation du code. Il s'agit de vérifier des caractéristiques telle que la sécurité ou la capacité à monter en charge. Les tests non-fonctionnels permettent plutôt de répondre à des questions telles que « est-ce que cette classe peut être utilisée par 1000 threads en même temps sans erreur ? ».

Tests unitaires, tests d'intégration et tests systèmes

Tandis qu'un test unitaire vise à tester une unité isolée pendant le test, un test d'intégration est un test qui met en œuvre plusieurs unités ou composants pour vérifier qu'ils fonctionnent bien ensemble. Le test système peut être considéré comme un test d'intégration global : il vérifie le fonctionnement de la totalité du système assemblé, lorsque tous les composants qui interviendront sont en place.

Isolation

Derrière la notion d'**isolation**, on distingue deux exigences. La première consiste à assurer l'isolation des composants testés, la seconde à assurer l'isolation des tests entre eux.

Isolation des composants testés

Il est important pour un test de ne tester *qu'une seule chose*, c'est à dire d'assurer le caractère *unitaire* du test. Supposons qu'il existe une classe `A`, son test unitaire est la classe `ATest`. De même, supposons maintenant qu'il existe une classe `B` (son test est `BTest`) qui dépend de `A`, c'est à dire qu'elle manipule des instances de A (appel de méthode) ou hérite de A. Si tous les tests passent, tout va bien.

Maintenant, supposez que le test de B ne passe pas. Il y a plusieurs hypothèses :

- il y a un bogue dans B ;
- il y a un bogue dans A, et cela provoque un comportement inattendu dans B.

Le problème auquel on est confronté est simple : Où le bogue se cache-t-il ? Faut-il chercher le bogue dans A ou dans B ? Pour être fixé, il faut repasser le test de A, s'il passe, c'est que le problème est dans B, sinon il est peut-être dans A ou ailleurs… En effet, nous avons réduit notre système à deux classes, mais que se passerait-il si A dépendait encore de plusieurs autres classes du système. On peut ainsi en introduisant un bogue, avoir des dizaines de tests qui ne passent plus dans diverses parties du système.

C'est typiquement un problème d'isolation. En théorie, un test unitaire ne doit tester qu'une *unité*, c'est à dire que s'il y avait un bogue dans A, seul le test de A devrait ne plus passer pour nous permettre de localiser précisément l'origine du problème. C'est dans le test de B que se situe le problème d'isolation : en effet, son succès dépend de la qualité de l'implémentation de A, alors que c'est B qu'on doit tester et pas autre chose. Le test de B ne devrait pas manipuler A mais utiliser des doublures de A, ceci afin de garantir l'isolation. En fait, notre test de B n'est pas un test unitaire mais plutôt un test d'intégration (qui vérifie que A et B interagissent bien).

Finalement, dans un système intégralement testé unitairement, chaque classe a son test, et chacun de ces tests ne manipule que la classe testée en isolation, en utilisant des doublures pour toutes les autres classes. Ainsi, si un test échoue, on sait exactement quelle est la classe où se trouve l'erreur de programmation.

Bien que séduisante, cette approche s'avère inapplicable. En effet, écrire autant de doublures pour tous ces tests devient contre-productif et s'avère pénible à la maintenance (chaque fois qu'un contrat change, il faut modifier toutes les doublures dans tous les tests des classes dépendantes). À vous de trouver le compromis entre deux extrêmes : une étanchéité complète entre les tests (beaucoup de code, temps de maintenance) et tests sans isolation (difficulté pour localiser l'origine du problème).

Isolation des tests entre eux

Pour illustrer la nécessité d'isoler les tests entre eux, prenons l'exemple du test d'un petit service `ServiceUser` proposant les opérations courantes (CRUD) pour gérer un base d'utilisateurs en permettant de les authentifier par mot de passe.

```
public class ServiceUserTest {

    /** Vérifie qu'on peut créer un utilisateur puis l'authentifier. */
    @Test
    public void testCreateConnectUser() {
        // on crée un utilisateur et on l'enregistre
```

```
        User user = new User();
        user.setLogin("toto");
        user.setPassword("mdp");
        serviceUser.createUser(user);

        // on doit pouvoir authentifier un utilisateur qui a été ajouté
        boolean authUser = serviceUser.connectUser("toto", "mdp");
        Assert.assertTrue(authUser);
    }

    /** Vérifie qu'on peut récupérer un utilisateur enregistré en base. */
    @Test
    public void testFindUser() {
        // on doit pouvoir retrouver un utilisateur enregistré
        // à partir de son login
        User user = serviceUser.findUserByLogin("toto");
        Assert.assertNotNull(user);
    }
}
```

Dans `testCreateConnectUser()`, le développeur crée un utilisateur en base et dans `testFindUser()`, il tente de le retrouver. Cette classe de test est mauvaise : le développeur suppose, pour le bon déroulement de `testFindUser()` que la méthode `testCreateConnectUser()` sera appelée avant. Cela est une mauvaise supposition pour trois raisons :

- D'abord, un développeur qui veut lancer seulement la méthode de test `testFindUser()` peut très bien le faire. JUnit, les IDE et les outils de build le permettent, cela fait partie du paradigme xUnit. Dans ce cas, la méthode `testCreateConnectUser()` n'est pas exécutée et l'utilisateur attendu n'a pas été ajouté en base donc le test échoue même si la classe ServiceUser est bien implémentée.
- Même si le développeur demande au test-runner de lancer tous les tests de la classe, rien ne garantie que les méthodes seront exécutées dans l'ordre dans lequel elles apparaissent dans le code source. Le test-runner peut très bien appeler `testCreateConnectUser()` *après* `testFindUser()`. Dans ce cas, `testFindUser()` échoue pour les mêmes raisons qu'au point précédent alors qu'il n'y a pas de bogue dans `ServiceUser`.
- Si le premier test se déroule mal, ce test termine et laisse le service dans un état inconnu, inattendu. Le second test peut alors échouer ou réussir aléatoirement, ce qui peut laisser supposer qu'il y a un bogue dans `findUserByLogin()` alors que ce n'est pas le cas.

Lorsque vous écrivez une méthode de test, vous devez toujours ignorer ce qui est fait dans les autres méthodes de test et ne jamais supposer qu'elles seront appelées avant ou après celle que vous écrivez.

Déterminisme

Un test n'est utile que s'il est reproductible. En effet, si un test détecte une erreur et qu'on a fait une correction, si on ne peut pas rejouer le test dans les mêmes conditions que celles qui ont produit l'échec, on ne pourra pas savoir si la correction est bonne ou si c'est un coup de chance.

C'est pourquoi il est nécessaire de fournir systématiquement en entrée du système testé le même ensemble de données. Sont donc à proscrire les données aléatoires (générées), les données issues de l'environnement hôte (date courante, locale, adresse réseau locale…). Pour assurer le déterminisme, il convient d'isoler le composant qui fournit ces données dans l'application pour en créer une doublure. Cette doublure fournira, selon le besoin pour les tests, une date courante fixée dans les tests, des nombres aléatoires générés à partir d'un générateur initialisé avec une graine fixe (fixer la graine du générateur rend déterministe la suite de nombre générée etc.). Les données de test générées peuvent également être enregistrées dans un fichier qui sera lu pour fournir les mêmes données à chaque exécution du test.

Il n'est pas convenable de laisser des tests non-déterministe dans la suite de tests. En effet, s'ils échouent, on ne peut vérifier si c'est à cause de l'indéterminisme ou si c'est l'insertion d'un bogue qui a provoqué l'échec[2].

Exhaustivité

La qualité de tests reposent évidemment sur leur exhaustivité, c'est-à-dire, si l'application est testée dans les différentes situations possibles de départ, les différentes utilisations qu'on peut faire du système, que le système fonctionne dans les cas nominaux et dans les cas d'erreurs (lorsqu'une ou plusieurs erreurs surviennent).

Lorsqu'on a une base de test conséquente, il devient difficile de distinguer ce qui est testé de ce qui ne l'est pas. Heureusement, des outils existent pour analyser notre suite de test et nous indiquer leur exhaustivité. L'approche la plus fréquemment retenue est le calcul de la *couverture* du code par les tests.

Couverture du code par les tests

La couverture du code (ou *code coverage*) permet d'évaluer la qualité d'un jeu de test en vérifiant quelles sont les parties du code qui sont appelées lors des tests.

Si du code est appelé pendant un test, cette portion de code est considéré comme couverte ; a contrario, tout le code non-appelé est considéré comme non-couvert. La couverture s'exprime donc sous forme d'un pourcentage représentant la proportion de code couverte sur la quantité totale de code.

Différentes façons de mesurer la couverture

On distingue notamment :

Couverture des méthodes (*function coverage* ou *method coverage*)
: qui vérifie que chaque méthode (publique, protégée ou privée) d'une classe a été exécutée.

Couverture des instructions (*statement coverage* ou *line coverage*)
: qui vérifie que chaque instruction a été exécutée.

Couverture des chemins d'exécution (*branch coverage*)
: qui vérifie que chaque parcours possible (par exemple, les 2 cas passant et non-passant d'une instruction conditionnelle) a été exécuté.

Cette différence est significative, prenons l'exemple suivant :

```
public void uneMethode(boolean test) {
    if (test) {
        instruction1();
    } else {
        instruction2();
        instruction3();
        instruction4();
        instruction5();
        instruction6();
        instruction7();
        instruction8();
        instruction9();
    }
}
```

Si votre test appelle cette méthode en passant `false` en paramètre, vous avez assuré la couverture en lignes

de 90% du code puisque vous avez exécuté 9 instructions sur les 10 présentes dans le corps de la méthode. Toutefois, vous n'assurez que 50% de la couverture en branche puisque une branche (le cas `test=true`) sur deux n'est pas testée.

Couverture d'efficacité des opérandes booléens (*boolean operand effectiveness* ou *MC/DC : Modified Condition/Decision Change*)

Ce type de couverture fait partie des types de couvertures pour le niveau le plus strict (niveau A) de la norme DO-178B utilisée en avionique et les secteurs où des systèmes critiques sont développés. La couverture consiste à couvrir plus en détails les expressions conditionnelles en mesurant l'efficacité de chacun des opérandes et permet la détection d'opérandes non effectif (jamais évalué, ou inutile dans l'expression car dépendant d'un autre).
Ce type de test est donc utilisé avec des expressions booléennes complexes dans les instructions de contrôle d'exécution (condition, boucle, ...). Les expressions booléennes sont en général évaluées de manière incomplète :

a OU b

si a est vrai, b n'a pas besoin d'être évalué car le résultat sera vrai quel que soit la valeur de b.

a ET b

si a est faux, b n'a pas besoin d'être évalué car le résultat sera faux quel que soit la valeur de b.

Pour comprendre ce type de couverture, voici un exemple d'expression conditionnelle pour tester si une année est bissextile ou non :

```
/* dans une fonction prenant un paramètre : annee  */
...
if ( (annee % 4 == 0) && ( (annee % 100 != 0) || (annee % 400 == 0) )
{
    ...
```

Chacun des opérandes doit être évalué à vrai et à faux sans que les autres opérandes ne changent, tout en changeant le résultat de la condition. L'expression contient 3 opérandes reliés par les opérateurs booléens OU (`||`) et ET (`&&`) :

- `(annee % 4 == 0)`
- `(annee % 100 != 0)`
- `(annee % 400 == 0)`

Les 4 tests suivants permettent une couverture d'efficacité des opérandes booléens de 100% : 1900, 1980, 1983, 2000. La mesure d'efficacité est résumée dans un tableau pour chacun des opérandes. Le point d'interrogation (?) signifie que l'opérande n'est pas évalué, et l'on peut considérer que ? == true ou ? == false.

Efficacité de `(annee % 4 == 0)`

Test annee =	`(annee % 4 == 0)`	`&&`	`( (annee % 100 != 0)`	`||`	`(annee % 400 == 0) )`	Résultat
1983	`false`		?		?	`false`
1980	`true`		`true`		?	`true`
Constat	**Changement**		**Constant**		**Constant**	**Changement**

Efficacité de `(annee % 100 != 0)`

Test annee =	`(annee % 4 == 0)`	`&&`	`( (annee % 100 != 0)`	`||`	`(annee % 400 == 0) )`	Résultat
1900	`true`		`false`		`false`	`false`
1980	`true`		`true`		?	`true`
Constat	**Constant**		**Changement**		**Constant**	**Changement**

Efficacité de `(annee % 400 == 0)`

Test annee =	`(annee % 4 == 0)`	`&&`	`( (annee % 100 != 0)`	`||`	`(annee % 400 == 0) )`	Résultat
1900	`true`		`false`		`false`	`false`
2000	`true`		`false`		`true`	`true`
Constat	**Constant**		**Constant**		**Changement**	**Changement**

Déterminer les cas de test pour cette couverture est plus complexe lorsque les opérandes sont des appels de fonctions ou méthodes.

Une métrique non-linéaire

S'il est facile, en écrivant des tests d'atteindre 50 % de couverture, il est plus difficile d'atteindre 70 à 80 %. Plus la couverture est grande, plus il est difficile de l'augmenter.

Selon les organisations et l'exigence de qualité, il peut être obligatoire d'avoir un pourcentage minimal de couverture. Voir, par exemple, la norme DO-178B (en avionique) qui exige, à partir de la criticité d'un composant la couverture en ligne, la couverture en branches ou les deux à 100% (tout écart devant être justifié dans un document).

Quelques outils pour évaluer les couvertures

À titre d'exemple, vous pouvez parcourir le rapport de couverture généré par Cobertura sur le framework Web Tapestry (http://tapestry.apache.org/current/tapestry-core/cobertura/) [archive](Lien mort). On peut y lire, pour chaque package et chaque classe la couverture en lignes et en branches ainsi qu'une mesure de la complexité du code. De plus, pour chaque classe, on peut voir, dans le détail, les lignes couvertes (surlignées en vert) et les lignes non-couvertes (surlignées en rouge). Autre exemple, le rapport de couverture généré par Emma sur la base de donnée H2 (http://h2database.com/coverage/overview.html) [archive].

Pour Java
: cobertura (http://cobertura.sourceforge.net/) [archive] (couvertures par lignes et par branches), emma (http://emma.sourceforge.net/) [archive] (eclemma (http://www.eclemma.org/) [archive] permet d'intégrer emma à Eclipse pour distinguer les lignes non-couvertes directement dans l'IDE).

Pour JavaScript
: script-cover (http://code.google.com/p/script-cover/) [archive], une extension pour le navigateur Google Chrome.

Pour PHP
: Xdebug (http://www.xdebug.org/docs/code_coverage) [archive]

Pour Python
: PyPi coverage (http://pypi.python.org/pypi/coverage) [archive], voir aussi cette conférence

(http://us.pycon.org/2009/conference/schedule/event/26/) [archive]

Pour Ruby

simplecov (https://github.com/colszowka/simplecov) [archive]

Qualité des tests par analyse de mutations

L'analyse de mutation (ou *mutation testing*) permet d'évaluer la qualité d'un test en vérifiant sa capacité a détecter les erreurs que le développeur aurait pu introduire.

Notion de « mutant »

Une classe mutante est une copie d'une classe originale dans laquelle on introduit une petite erreur. Parmi ces erreurs, on peut :

- supprimer une instruction ;
- inverser l'ordre de deux instructions ;
- remplacer tout ou partie d'une expression booléenne par `true` ou `false` ;
- remplacer des opérateurs (par exemple, remplacer == par `!=` ou <= par >) ;
- remplacer une variable par une autre.

Principe de l'analyse de mutation

Pour une classe C donnée et sa classe de test T :

1. On génère tous les mutants ;
2. Pour chaque mutant C', on fait passer le test T ;
   - si le T échoue, le mutant est éliminé ;
   - si le T passe, le mutant est conservé.

Un bon test doit éliminer tous les mutants.

Limites

Le principal défaut de ce système est qu'il est probable qu'un mutant soit équivalent à la classe originale (le code diffère syntaxiquement, mais la sémantique du code est identique). Comme il est équivalent, il n'est pas éliminé et provoque un faux-positif. Cependant, le test est bon même si le mutant n'est pas éliminé.

Outils

Java

PIT (http://pitest.org/) [archive]

Qualité du code

Au même titre que le code de l'application elle-même, le code des tests doit être compréhensible par tous et maintenu sur le long terme. Il doit donc être développé avec la même rigueur que du code habituel : respect des conventions, éviter le code copié-collé (factoriser le code commun dans une classe mère, une classe utilitaire ou une classe externe), utilisation parcimonieuse des patrons de conception et de l'héritage, commentaires, documentation, etc.

L'effort doit être mis sur la simplicité et l'évidence de code des tests. Dès lors qu'on a un doute sur

l'exactitude d'un test (on se demande, lorsque le test passe, si c'est parce que le code est bon ou le test trop faible), la démarche de test devient absurde. L'extrême limite est franchie lorsque qu'un test est tellement compliqué qu'il devient nécessaire de tester le test…

Références

1. il peut aussi s'agir d'une user story
2. À ce sujet, voir l'article « Eradicating Non-Determinism in Tests », par Martin Fowler (http://martinfowler.com/articles/nonDeterminism.html) [archive]

Intégration dans le processus de développement

Jusqu'ici, nous avons vu comment vous pouviez développer dans votre coin des tests portant sur le code que vous écrivez. Cependant, le développement d'un logiciel est un processus plus global : voyons comment les tests s'intègre avec les autres tâches à effectuer et comment on peut travailler en équipe.

Développer avec les tests

Bien que ce ne soit pas systématique, la pratique courante est que le code et les tests sont écrits par la même personne.

Écrire des tests prend du temps qu'on pourrait passer à écrire du code apportant de nouvelles fonctionnalités. Cela peut paraître frustrant, mais il faut renoncer à la course à la fonctionnalité pour préférer avancer plus lentement mais plus sûrement. En particulier, on est tenté de négliger l'écriture des tests après plusieurs semaines passées sur un projet, surtout en phase de bouclage où « on a plus le temps pour ça ». Appliquer une démarche de tests rigoureuse tout au long du projet permet de pouvoir modifier le code aussi sereinement en fin de projet qu'au début (les tests assurant la non-régression). La solution réside dans l'équilibre, c'est en fonction des contraintes du projet et du sentiment de maîtrise des développeurs qu'il convient de placer le curseur entre deux extrêmes qu'il faut éviter : « pas de temps pour les tests » et « 100 % de couverture, peu importe le temps que ça prendra ! ».

Après plusieurs mois de pratique, un développeur peut se demander « À quoi bon écrire des tests, on ne trouve jamais de bogues ». On peut le comprendre étant donné qu'après la mise en place d'une démarche des tests, les bogues se raréfient : on peut alors avoir l'impression que les tests ne servent à rien. Ce serait se tromper. Pour s'en convaincre, il suffit d'abandonner les tests pour voir ressurgir des bogues à retardement.

Les tests et la gestion de version

Si vous travaillez à plusieurs sur le développement du projet, vous utilisez probablement un outil de gestion de version de code-source (comme CVS ou Subversion). Vous pouvez l'utiliser pour stocker vos classes de tests aux côtés des classes.

Vous pouvez adopter des règles strictes :

- Ne jamais envoyer une classe qui n'est pas accompagnée d'un test
- Toujours faire passer les tests lorsqu'on récupère un projet
- Avant de faire un commit :
    1. Relancer tous les tests
    2. Mettre à jour le code

3. Repasser tous les tests
4. Faire le commit

Le développement piloté par les tests

Le développement piloté par les tests (ou « Test-Driven Development » ou « TDD ») est une pratique souvent utilisée dans les méthodes agiles (on trouve son origine dans l'extreme programming). Elle consiste à développer l'application selon le cycle suivant :

1. Écrire les tests
2. Vérifier que ceux-ci ne passent pas
3. Écrire le code manquant
4. Vérifier que le test passe
5. Remanier le code

1. Écrire les tests

Dans le développement piloté par les tests, les tests sont écrit avant le code. Il faut donc commencer par écrire un test ou un petit ensemble de tests qui représente les nouvelles fonctionnalités qu'on va implémenter durant le cycle.

Il faut pour cela se baser sur la spécification, les cas d'utilisation ou les *user stories*. Écrire les tests d'abord permet au développeur de voir vraiment le comportement qui est attendu avant de toucher au code.

2. Vérifier que les nouveaux tests échouent

Il est important de lancer les tests pour s'assurer que les autres passent toujours et que les nouveaux tests ne passe pas. Dans le cas contraire, deux possibilités :

- Le test n'est pas bon
- La fonctionnalité est déjà implémentée. Cela peut arriver étant donné qu'un développeur zélé peut, lorsqu'il touche une partie du code, ajouter dans la foulée un petit peu de plus de fonctionnalité que prévu.

Toutefois, si ce dernier cas se présente plusieurs fois, il faut se poser des questions sur la gestion du projet. Les tâches ont-elles bien été réparties ?

3. Écrire le code

Il est important d'essayer de n'écrire strictement que le code nécessaire au passage du test, pas plus. Vous pouvez relancer le test écrit à l'étape 1 autant de fois que nécessaire. Peu importe si le code n'est pas élégant pour l'instant tant qu'il permet de passer le test.

4. Vérifier que tous les tests passent

C'est le moment de vérifier que tous les tests passent, ceci afin de vérifier qu'avec les modifications faites on a pas créer de régressions dans le code.

5. Remanier le code

Enfin, *refactorisez* le code pour améliorer la conception tout en vérifiant que les tests passent toujours.

Intégrer les tests à la construction automatique du projet

Avec Ant

Si vous utilisez Ant pour construire vos projets, il existe la tâche `junit`. Attention, elle dépend des fichiers junit.jar et ant-junit.jar qu'il faut télécharger et indiquer dans le fichier build.xml. Référez vous à la documentation de la tâche `junit` dans la documentation officielle de Ant (http://ant.apache.org/manual/OptionalTasks/junit.html) [archive].

Avec Maven

Si vous utilisez maven pour construire et gérer les dépendances de votre projet. Vous pouvez lui demander de faire passer tous les tests.

```
mvn test
```

Maven intègre une convention pour placer les tests dans l'arborescence du projet. Vous pouvez déclarer JUnit et vos autres outils comme des dépendances. Vous pouvez également intégrer à la génération du site, la génération et la publication des rapports sur le respect de conventions et la couverture.

Gardez l'essentiel des tests rapidement exécutables

Quoi qu'il en soit, la construction du projet doit rester légère, et dérouler les tests ne doit pas prendre plus de quelques minutes. Le risque, en laissant le temps de construction s'allonger indéfiniment au fur et à mesure que les tests s'accumulent, est de lasser les développeurs qui seront tentés de sauter la phase de test de la construction.

Une bonne pratique consiste à laisser les tests unitaires dans la construction du projet et à ignorer les tests d'intégration dans le build par défaut. Une petite option (un profil Maven) permettant d'activer le passage des tests d'intégration. Le serveur d'intégration, lui, pourra construire systématiquement en repassant tous les tests.

Profitez des ressources du serveur d'intégration continue

Si vous faites de l'intégration continue, vous utilisez un système de gestion de version et vous avez sûrement un serveur chargé de construire régulièrement le projet. Vous pouvez faire en sorte que chaque construction rejoue l'ensemble des tests pour vérifier que tout va bien. Cette construction automatique régulière peut donner lieu à la génération de rapports. Votre serveur d'intégration peut donc vous permettre de suivre l'évolution des métriques et de la complexité du projet.

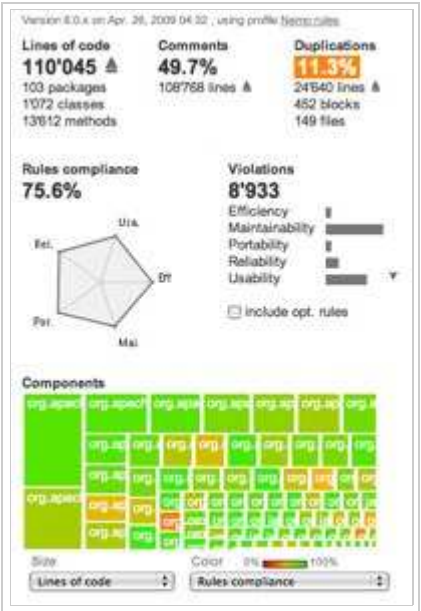
Coupler le serveur d'intégration continu avec un outils de *panneau de contrôle* tel que Sonar (http://sonar.codehaus.org/) [archive] vous permettra de suivre l'évolution du succès des tests et de la couverture du code en fonction du temps et de l'évolution de la complexité de l'application. Sonar permet de voir où se trouvent les portions de codes les moins testées.

Rapprocher les tests de l'utilisateur

Si les méthodes agiles recommandent les tests, elle recommandent également d'intégrer l'utilisateur au

processus de développement afin que les réalisations des développeurs collent au plus près des véritables attentes qu'on peut avoir du logiciel.

Les tests sont d'abord une discipline technique mais nous allons voir que par différentes façon, nous pouvons rapprocher ces travaux des utilisateurs. Les trois approches suivantes tentent de mêler les tests avec la documentation utilisateur (le cas doctest) ou avec les spécifications du logiciel (Fit et BDD). L'objectif est d'avoir des tests écrit dans un langage compréhensible par l'utilisateur, voire de permettre à l'utilisateur d'écrire les tests lui-même sous forme de tableau de valeurs dans un tableur (Fit) ou en langage simili-naturel (BDD). Idéalement, ces technologies permettrait de confronter l'implémentation du logiciel aux spécifications attendues, décrites en langue naturelle : on peut parler de *spécification exécutable* (anglais *executable spec*).



Sonar donne une vue d'ensemble de l'évolution des métriques d'un projet

Mêler tests et documentation, l'approche *doctest*

doctest est un outil livré avec Python qui permet, dans une documentation au format texte brut ou rST, d'ajouter des lignes qui permettent de vérifier ce qui vient d'être dit. L'exemple suivant montre un fichier texte d'exemple, il pourrait s'agir d'un fichier `README.txt`.

```
======================
Demonstration doctests
======================

This is just an example of what a README text looks like that can be used with
the doctest.DocFileSuite() function from Python's doctest module.

Normally, the README file would explain the API of the module, like this:

   >>> a = 1
   >>> b = 2
   >>> a + b
   3

Notice, that we just demonstrated how to add two numbers in Python, and
what the result will look like.
```

En lisant ce fichier, doctest (http://docs.python.org/library/doctest.html) [archive] va interpréter les lignes préfixées par >>> et vérifier que ce qui est retourné par l'évaluation de l'expression correspond à ce qui est écrit (ici, doctest va vérifier que l'évaluation de $a + b$ renvoie bien 3).

Des tests orientés données, l'approche Fit

Ward Cunningham propose dans son outil « fit » de rédiger les tests de validation dans un document, sans code. Son outil comprend un lanceur de test mais aussi un wiki embarqué, accessible via une interface Web. Dans ce wiki, les utilisateurs peuvent entrer, dans des tableaux, les jeux de données à fournir en entrée ainsi que les résultats attendus en sortie. Une autre possibilité, donnée aux utilisateurs, est d'entrer les jeux de données dans un tableur.

Java

Fit (http://fit.c2.com/) [archive], FitNesse (http://fitnesse.org/) [archive] (avec trinidad (http://code.google.com/p/trinidad/) [archive]), Concordion (http://www.concordion.org/) [archive]

PHP

phpfit (http://developer.berlios.de/projects/phpfit/) [archive]

Des tests fonctionnels en phase avec les attentes de l'utilisateur, l'approche *BDD*

C'est en essayant d'enseigner l'approche TDD que Dan North s'est rendu compte que la plupart des développeurs acceptent volontiers d'écrire les tests avant le code mais se contentent de percevoir cette technique comme une façon de tester en même temps qu'on développe, sans percevoir ce qu'il s'agit en fait d'une méthode de développement. Pour faciliter cette compréhension de TDD, Dan North a étendu les notions de bases pour former Behavior Driven Development, une méthode sensée apportée aux développeurs tous les bénéfices d'une véritable approche de TDD.

Des tests bien nommés

Notion de « Comportement »

Les attentes utilisateurs sont aussi des comportements

Frameworks de BDD

Groovy
: GSpec (http://groovy.codehaus.org/Using+GSpec+with+Groovy) [archive], Spock (http://spockframework.org/) [archive], easyb (http://www.easyb.org/) [archive]

Java
: Cucumber-JVM, un port officiel de Cucumber pour les différents langages de la JVM (http://github.com/cucumber/cucumber-jvm) [archive], Instinct (http://code.google.com/p/instinct/) [archive], JBee (http://sites.google.com/site/jbeetest/) [archive], JBehave (http://jbehave.org/) [archive], JDave (http://www.jdave.org/) [archive], Robot Framework (http://code.google.com/p/robotframework/) [archive], Narrative (http://youdevise.github.com/narrative/) [archive]

JavaScript
: Jasmine (http://pivotal.github.com/jasmine/) [archive]

PHP
: behat (http://behat.org/) [archive]

Python
: Freshen (http://github.com/rlisagor/freshen) [archive], Lettuce (http://lettuce.it) [archive], Pyccuracy (http://www.pyccuracy.org/) [archive], Pyhistorian (http://github.com/hugobr/pyhistorian) [archive], PyCukes (http://github.com/hugobr/pycukes) [archive]

Ruby
: RSpec (http://rspec.info/) [archive], Cucumber (http://cukes.info/) [archive]

Scala
: ScalaTest (http://www.scalatest.org/) [archive], specs (http://code.google.com/p/specs/) [archive]

Pour aller plus loin

- L'article « Introducing BDD », par Dan North (http://dannorth.net/introducing-bdd/) [archive]
- La page « Introduction » sur behavior-driven.org (http://behaviour-driven.org/Introduction) [archive]

Tests de non-régression

Il s'agit de tests qui visent à assurer la non-régression d'une application. C'est à dire qu'une modification

apportée à l'application ne rend pas erroné les comportements d'une application qui fonctionnaient avant la modification.

« Beta test »

Le beta-test consiste à fournir une version quasi-finale de l'application (dite « version Beta ») à un large échantillon d'utilisateurs finaux. Ceci afin qu'ils rapportent les derniers bogues résiduels ou dans le cas d'une interface graphique, quelques améliorations ergonomique. La version livrée est estampillée « Beta » afin que les utilisateurs soient conscient qu'il ne s'agit pas le version finale et que le version présentée peut encore contenir des erreurs.

On parle de bêta ouverte ou de bêta fermée selon que cette version du logicielle soit accessible à tous les utilisateurs potentiels ou seulement à un groupe restreint.

Problématiques de tests

Ce chapitre vise à décrire des approches pour résoudre des problématiques de tests récurrentes et présente des outils complémentaires qui répondent à ces problématiques spécifiques.

Tester des applications concurrentes


Cette section est vide, pas assez détaillée ou incomplète.

Faire appel à la concurrence dans une application peut être nécessaire, souvent pour des raisons de performance. Toutefois, la multiplication des processus, des threads ou des fibres peut créer des bugs (interblocage, famine…) car il est difficile de bien synchroniser l'application. De plus, l'ordonnancement des tâches par les systèmes d'exploitation multi-tâches ou sur les systèmes parallèles (processeurs multi-cœurs ou systèmes multi-processeurs) n'est pas toujours le même à chaque exécution de l'application ou des applications, ce qui complique les tests.

Tout d'abord, il convient de mener une analyse statique du code de l'application, cela devrait éliminer les erreurs fréquentes.

w:en:Concutest ?

Tester des applications distribuées

Qu'il s'agisse d'applications client-serveurs, d'une application en architecture n-tiers, ou d'un système distribué en grande échelle (tel qu'un système pair-à-pair) : ces systèmes sont intrinsèquement concurrents. En plus de l'aspect distribué, ils posent donc aussi bien la problématique du test de système concurrent.

Une des solutions consiste, via un outil spécialisé, à faire intervenir, pour tous les nœuds un seul et même *TestRunner* centralisé faisant appel aux différents tests les uns après les autres, en attendant que chaque machine soit synchrone par rapport au déroulement du test général. Cette synchronisation centralisée lève l'indéterminisme induit par la distribution et permet de tester normalement. C'est l'approche retenue par deux outils :

JBoss Distributed Testing Tool (http://www.jboss.org/jbossdtf) [archive]

…

PeerUnit (http://peerunit.gforge.inria.fr/) [archive]

Un projet INRIA plutôt destiné au test des systèmes pair-à-pairs.

Cette approche à l'avantage de permettre de tester un système *in situ*, vraiment distribué (avec les temps de latence réseau notamment). Principal inconvénient, pour automatiser le déroulement des tests, il faudra également automatiser le déploiement.

Tester une application avec des données persistantes

http://www.dbunit.org/

Les « fixtures »

Lorsque le système testé exploite une base de données, l'écriture des tests devient compliquée. En effet, si on utilise une base de données de test présente sur la machine du développeur, les données risquent de changer au cours du temps (tests manuels). De même, si on exécute plusieurs tests de suite sur une même base de données et qu'un test qui modifie les données en base échoue, on a laissé la base dans un état indéterminé, peut-être incohérent, ce qui peut invalider le reste des tests.

C'est pourquoi il faut absolument cloisonner les tests et, avant chaque test, remettre la base dans un état cohérent et déterminé. Cela peut être laborieux et cela peut aboutir à des tests qui prennent 100 lignes pour insérer des données en base (avec le risque d'erreur) puis 5 lignes pour faire effectivement le test.

C'est ici que les « fixtures » interviennent : ces *fixtures* sont des doublures qui vont venir peupler la base (vidée entre chaque test) pour la placer dans un état cohérent juste avant le test.

On décrit ce petit jeu de données dans un fichier lisible, souvent au format YAML (plus rarement en XML ou en JSON). Par exemple :

```
# Fixtures pour la table 'books'
miserables:
  title: Les misérables
  author: Victor Hugo
  year: 1862

lotr:
  title: The Lord of the Rings
  author: J. R. R. Tolkien
  year: 1954

dunces:
  title: A Confederacy of Dunces
  author: John Kennedy Toole
  year: 1980
```

À chaque modification du schéma de la base, on crée ou on adapte ces petits fichiers. Une API permet alors d'interpréter ces fichiers et de charger les données en base depuis les tests. Par exemple, avec le framework Play! (Java) :

```
@Before
public void setUp() {
    Fixtures.deleteAll();
    Fixtures.load("data.yml");
```

```
}
```

En l'état actuel, il faut reconnaître qu'il n'existe pas ou peu d'outils indépendants pour les fixtures. Ces derniers sont plutôt fournis avec les frameworks Web comme c'est le cas dans Play! (Java) (http://www.playframework.org/documentation/1.1/test#fixtures) [archive], Django (Python) (http://docs.djangoproject.com/en/dev/topics/testing/#topics-testing-fixtures) [archive], Ruby on Rails (http://api.rubyonrails.org/classes/Fixtures.html) [archive] ou CakePHP (http://book.cakephp.org/fr/view/358/Preparing-test-data) [archive]. Cela est bien dommage, l'usage des bases de données ne se limitant pas aux applications Web.

« Fixtures » se traduirait par « garniture », mais il semble plus pertinent d'utiliser le terme « échantillons ».

Tester les procédures embarquées d'une base de données

Pour des raisons de performances, il est possible que des traitements *métiers* sur les données soient réalisés en base. Ces traitements sont des codes stockés directement dans le SGBD, ils sont codés dans un langage tels que PL/SQL ou PL/pgSQL.

Plutôt que de tester ces procédures au milieu des autres tests de l'application, il est préférable de mener les tests au plus près, avec des outils insérant les données et déclenchant les procédures. De simples requêtes de lecture en base et des assertions permettent de s'assurer que les traitements effectués par les procédures stockées produisent de bons résultats.

Parmi ces outils, on peut citer SQLUnit (http://sqlunit.sourceforge.net/) [archive].

Résoudre des dépendances vers des composants complexes

S'il est possible d'écrire des doublures pour des petits composants, certains composants complexes ne peuvent pas être vraiment doublés. Par exemple, une base de données ou un serveur de mail (SMTP) pour une application qui envoie des e-mails à ses utilisateur.

Dans ce cas, il faut plutôt essayer de remplacer ces composants usuels par d'autres briques plus adaptées, voire prévues pour les tests.

Bases de données

Il est fort probable que l'application a vocation à être utilisée, en production, sur une base de données client/serveur (MySQL, PostgreSQL ou autre). Cependant, on peut envisager de profiter d'une couche d'abstraction de la base de données[1] pour substituer cette base client/serveur par une base de données embarquée (comme SQLite pour PHP/Python/Ruby ou HSQLDB pour Java). Bien que ces bases de données soient moins performantes et ne gèrent souvent pas la concurrence ou d'autres fonctionnalités attendues d'une bonne base de données, elles sont très souvent amplement suffisantes pour travailler sur un petit ensemble de données de test.

Serveur de mail

L'application peut avoir besoin d'envoyer des courriels automatiques à ses utilisateurs (notifications, confirmations…). Sur leurs machines, les développeurs, lors des tests peuvent substituer au serveur de mail (SMTP) un simulacre comme Wiser (http://code.google.com/p/subethasmtp/wiki/Wiser) [archive] pour Java ou fakemail (http://www.lastcraft.com/fakemail.php) [archive] pour Python (utilisable aussi avec SimpleTest pour PHP). Ces simulacres s'installent et se mettent à l'écoute du port 25, pour recevoir les requêtes SMTP envoyées par l'application.

Ces solutions permettent d'utiliser l'application et de provoquer l'envoi de courriels sans risquer de vraiment envoyer les courriels. Par contre, ces outils permettent de vérifier qu'un courriel a bien été envoyé. Illustrons cela à l'aide d'un extrait de la documentation de Wiser :

```
Wiser wiser = new Wiser();
wiser.setPort(2500); // Default is 25
wiser.start();

// Après envoi du courrier

for (WiserMessage message : wiser.getMessages()) {
    String envelopeSender = message.getEnvelopeSender();
    String envelopeReceiver = message.getEnvelopeReceiver();
    MimeMessage mess = message.getMimeMessage();

    // il n'y a plus qu'à vérifier que ces messages ont le bon
    // destinataire, le bon contenu…
}
```

Système de fichiers

Pour tester des applications qui font appel au stockage de données sur le système de fichiers (partage de fichiers en réseau, logiciels de sauvegarde, de téléchargement ou de transferts de fichiers…), il peut être utile d'avoir une doublure pour simuler le système de fichiers et pouvoir faire des vérifications sur les données lues ou écrites.

Ruby
: fakefs (https://github.com/defunkt/fakefs) [archive]

Java
: MockFtpServer (http://mockftpserver.sourceforge.net/) [archive]

GPS

Python
: gpsfake (http://gpsd.berlios.de/gpsfake.html) [archive]

Tester des interfaces graphiques

Le monkey testing

Un « monkey test » (littéralement *test du singe*) est en fait un test de fuzzing appliqué aux interfaces graphiques. Le singe va lancer l'interface, et appuyer au hasard sur des boutons, entrer des données aléatoires etc. On peut ainsi détecter les cas où une mauvaise entrée provoque une erreur.

Les développeurs Android (une plateforme pour téléphone mobile qui permet de réaliser des applications graphiques prenant en charge un écran tactile) peuvent utiliser un singe fourni avec le Kit de développement (http://developer.android.com/guide/developing/tools/monkey.html) [archive] pour tester leurs applications. Le singe appuie sur des boutons, remplit les formulaires mais peut aussi simuler des glisser-déposer sur l'écran tactile.

Tester des interfaces Web

Divers outils spécialisés permettent de programmer des cas d'utilisation d'une interface Web, par exemple :

1. Charger cette URL,
2. Remplir un formulaire,
3. Valider le formulaire,
4. Vérifier que la page chargée contient bien un texte donné.

L'outil peut également proposer, plutôt que de programmer les tests, d'enregistrer la série de manipulations effectuées par un utilisateur dans un navigateur (clics, soumission de formulaires...) et de la reproduire dans les tests.

Parmi ces outils, citons :

watir (http://watir.com/) [archive]
: qui s'adresse plutôt au développeurs Ruby. Watij (http://watij.com/) [archive] est un portage Java

Windmill (http://www.getwindmill.com/) [archive]
: qui propose des API Python, JavaScript et Ruby

Selenium HQ (http://seleniumhq.org/) [archive]
: qui propose des API C#, Java, Perl, PHP, Python et Ruby

Canoo WebTest (http://webtest.canoo.com/) [archive]
: permet de décrire les tests en XML ou en Groovy ou de les enregistrer depuis Firefox avec WebTestRecorder (http://webtestrecorder.canoo.com/) [archive]

Ces outils sont comparés dans l'article « List of web testing tools » de Wikipedia anglophone.

Tellurium Automated Testing Framework (http://code.google.com/p/aost/) [archive], JWebUnit (http://jwebunit.sourceforge.net/) [archive].

Tester des interfaces lourdes

De la même façon, des outils sont spécialisés pour le test fonctionnel d'interfaces utilisateurs lourdes basées sur toolkit comme SWING (Java), par exemple.

Python
: dogtail (https://fedorahosted.org/dogtail/) [archive]

Tester du code destiné à fonctionner dans un serveur d'application


Cette section est vide, pas assez détaillée ou incomplète.

En Java, si vous développez une application faisant appel à des composants JEE, tels qu'EJB, JPA ou autre, vous n'allez pas tester en déployant votre application dans un JBoss durant les tests. Vous pouvez essayer de remplacer ces dépendances par d'autres plus légères (comme Apache OpenEJB (http://openejb.apache.org/) [archive]). Depuis Java 1.6, il doit être possible d'utiliser EJB de façon embarquée via javax.ejb.embeddable.* (http://docs.oracle.com/javaee/6/api/javax/ejb/embeddable/package-summary.html) [archive].

Java
: Cactus (http://jakarta.apache.org/cactus/) [archive], JBoss Arquillian (http://www.jboss.org/arquillian) [archive]

Tester une application Android

Cette section est vide, pas assez détaillée ou incomplète.

L'API Android intègre déjà des éléments permettant d'écrire des tests pour les activités, ils se trouvent dans package android.test.* (http://developer.android.com/reference/android/test/package-summary.html) [archive]. Un outils de doublures est également inclus dans package android.test.mock.* (http://developer.android.com/reference/android/test/mock/package-summary.html) [archive]. La documentation développeur intègre plusieurs chapitres consacrés au tests (http://developer.android.com/guide/topics/testing/index.html) [archive].

Il existe également des outils spécialisés :

- Robolectric (http://pivotal.github.com/robolectric/) [archive]
- Robotium (http://code.google.com/p/robotium/) [archive]

Gérer du code patrimonial

Le code patrimonial (« code légué » ou « legacy code ») est une base de code souvent de mauvaise qualité, dont plus personne ou peu de personnes n'a encore la connaissance. Il s'agit de projets anciens faisant souvent appel à des technologies anciennes voire obsolètes mais qui sont toujours en production.

Les tests peuvent aider à maintenir ce genre de code. Lorsqu'on doit ainsi corriger un bogue dans une telle application et qu'il n'y a pas ou peu de tests, on peut commencer par écrire des tests, simples d'abord. On poursuit ensuite en raffinant les tests, pour se rapprocher de l'origine du bogue. On a trouvé le bogue une fois qu'on a écrit un test qui ne passe pas alors qu'il devrait. Le bogue, ainsi isolé, peut maintenant être corrigé. On peut repasser tous les tests écrits depuis le début pour vérifier qu'on a pas provoquer une régression en modifiant le code. On parle alors de « characterization test », des tests qui assurent que le programme fonctionne de la même manière qu'au moment où les tests ont été écrits.

En procédant ainsi pour chaque bogue découvert, on constitue une base de tests et on tend ainsi à retrouver un environnement de travail plus sain.

Références

1. Il peut s'agir d'une simple couche d'abstraction au niveau de SQL (JDBC pour Java, PDO pour PHP…) qui permet d'interagir avec une base de données selon une API commune. Il suffit alors de changer le *pilote* chargé au lancement de l'application pour l'adaptée au SGBD utilisé. Un pilote pour MySQL, un autre pour SQLite, etc. : il suffit de changer le pilote pour changer de SGBD et le code reste inchangé. Une autre couche, plus haut niveau, peut permettre de s'abstraire du SGBD utilisé. Il s'agit de la couche de mapping objet-relationnel : Hibernate pour Java, SQLAlchemy pour Python, Active record pour Ruby, Doctrine pour PHP… Toutefois la plupart de ces couches de mapping reposent sur la première couche que nous avons décrite, cela reste donc une question de changement de pilote : une simple directive de configuration.

Conclusion

Dans la littérature qui touche aux doublures et à leurs utilisations, il y a des inconsistances sur la nomenclature : il ne faudra pas s'étonner de lire des documents qui utilisent des termes différents ou inversent les termes que nous avons utilisé dans le chapitre consacré aux doublures [1].

Ressources sur le Web

- Le portail « Software Testing » de Wikipédia anglophone ;
- les question-réponses sur les tests sur Stack Overflow (http://stackoverflow.com/questions/tagged/testing?sort=votes) [archive], notamment :
  - « Why didn't unit testing work out for your project? » (http://stackoverflow.com/questions/301693/why-didnt-unit-testing-work-out-for-your-project) [archive] ;
- les cours consacrés aux tests sur developpez.com (http://conception.developpez.com/cours/?page=qualite-cat#tests) [archive] (**attention, peu de documents sont à jour**, vérifiez les dates) ;
- Plusieurs articles intéressants :
  - TDD Anti-patterns (http://blog.james-carr.org/2006/11/03/tdd-anti-patterns/) [archive] : James Carr, présente en 2006 une liste d'*anti-patron* de tests qui révèle une mauvaise pratique de tests ou une mauvaise conception de l'application testée ;
  - Mocks Aren't Stubs (http://martinfowler.com/articles/mocksArentStubs.html) [archive], de Martin Fowler.

Actualités

- Le blog « The green bar », par David Saff ingénieur-testeur chez Google et *commiteur* JUnit (http://saffgreenbar.blogspot.com/) [archive] ;
- Google Testing Blog (http://googletesting.blogspot.com/) [archive]
- Les vidéos des conférences de la Google Test Automation Conference (GTAC) (http://www.gtac.biz/videos) [archive] : 2007 (http://www.youtube.com/results?search_query=GTAC2007&search=tag) [archive], 2008 (http://www.youtube.com/results?search_query=GTAC2008&search=tag) [archive], 2009 (http://www.youtube.com/results?search_query=GTAC2009&search=tag) [archive], 2010 (http://www.youtube.com/results?search_query=GTAC2010&search=tag) [archive], 2011 (http://www.youtube.com/results?search_query=gtac%202011&search=tag) [archive]
- Les vidéos des conférences sur Software Testing & Quality Assurance Videos Directory (http://www.testingtv.com/) [archive].

Ressources pour chaque langage

Groovy

- Testing Guide (http://groovy.codehaus.org/Testing+Guide) [archive]

Java

- le chapitre « Tests » du wikilivre « Développer en Java »

JavaScript

- Javascript Unit-testing? (http://stackoverflow.com/questions/32809/javascript-unit-testing) [archive] aborde le sujet des frameworks xUnit pour JavaScript

Python

- le « chapitre consacré aux tests unitaires de « Plongez au cœur de Python » » (http://diveintopython.adrahon.org/unit_testing/) (*Archive (https://web.archive.org/web/*/http:*

//diveintopython.adrahon.org/unit_testing/) [archive] • *Wikiwix (https://archive.wikiwix.com/cache/?url=http://diveintopython.adrahon.org/unit_testing/)* [archive] • *Que faire ?*);

- Une « série d'article publiés sur le blog Nic0's Sphere » (http://www.nicosphere.net/series/test-unitaire-python/) (*Archive (https://web.archive.org/web/*/http://www.nicosphere.net/series/test-unitaire-python/)* [archive] • *Wikiwix (https://archive.wikiwix.com/cache/?url=http://www.nicosphere.net/series/test-unitaire-python/)* [archive] • *Que faire ?*).

Ruby

- l'article « Unit-testing frameworks for Ruby » sur Wikipedia anglophone
- le chapitre « Unit Testing » du wikibook « Programming Ruby», « A Guide to Testing Rails Applications » (http://guides.rubyonrails.org/testing.html) [archive]

Scala

- Les tests dans « Programming Scala » (http://ofps.oreilly.com/titles/9780596155957/ScalaToolsLibs.html#_test_driven_development_in_scala) [archive]

Bibliographie

- (anglais) **Gerard Meszaros**, *XUnit Test Patterns: Refactoring test codes*, [Professional (http://www.pearsonhighered.com/%7CAddison-Wesley) [archive]], 21 mai 2007 (ISBN 978-0-13-149505-0) (ancienne édition disponible en ligne) (http://xunitpatterns.com/) [archive]

Références

1. Ce chapitre est cohérent avec la nomenclature adoptée par Martin Fowler (http://martinfowler.com/bliki/TestDouble.html) [archive].