Tests unitaires et doublures de tests : les simulacres ne sont pas des bouchons

par Martin Fowler (Site Web de Martin Fowler) (traduction par Bruno Orsier)

Date de publication : 31/8/2007

Dernière mise à jour : 29/10/2007

Le terme "objet simulacre" est devenu populaire pour décrire des objets spéciaux qui imitent de vrais objets dans le but de les tester. La plupart des environnements de développement ont maintenant des outils qui permettent de créer facilement des objets simulacres. Cependant, souvent on ne réalise pas que les objets simulacres sont un cas particulier d'objets de tests, qui permettent un style de test différent. Dans cet article j'explique comment les objets simulacres fonctionnent, comment ils encouragent le test basé sur la vérification du comportement, et comment la communauté autour d'eux les utilise pour développer un style de test différent.
Dernière mise à jour significative : 02 Janvier 2007

Notes du traducteur

Le papier original en anglais est disponible **ici**.

Les exemples de cet article sont en Java, mais je les ai **convertis en C#**, (en utilisant par exemple NMock2, TypeMock) afin de rendre cet article plus facilement accessible à la communauté .NET.

Le vocabulaire anglais employé dans la littérature de test est assez délicat à traduire en français (et je suis d'ailleurs soulagé que Martin Fowler souligne que même en anglais ce vocabulaire est assez confus). J'ai suivi les traductions proposées **ici**. Donc la traduction utilise la table de correspondance ci-dessous :

mot anglais	mot français
mock	simulacre
fake	substitut
stub	bouchon
dummy	fantôme

Il y a plusieurs alternatives : **objets fantaisie**, mimes, etc.

Le mot *stub* est particulièrement vide de sens pour un francophone... Cette **discussion** au sujet de la traduction de *to stub out* est assez révélatrice.

En ce qui concerne la traduction de l'acronyme TDD (pour Test Driven Development), l'équivalent français DDT (Développement Dirigé par les Tests) n'est pas très parlant, ni très courant. Par conséquent TDD n'a pas été traduit. De même pour BDD (Behavior Driven Development) dont l'équivalent français serait DDC (Développement Dirigé par le Comportement). BDD n'a pas été traduit non plus.

Remerciements

Merci à **Miles**, **Ricky81**, **UNi[FR]**, **RideKick** pour leurs relectures de cette traduction, et à **Alp** pour sa relecture des exemples en C#.

Merci également à **Ditch** et **Giovanny Temgoua** pour leurs diverses suggestions qui ont permis d'améliorer cet article.

Introduction

J'ai rencontré pour la première fois le terme "objet simulacre" dans la communauté XP il y a quelques années. Depuis je les ai rencontrés de plus en plus. D'une part parce ce que beaucoup des principaux développeurs de ces objets ont fait partie de mes collègues à ThoughtWorks à différents moments. D'autre part parce que je les vois de plus en plus dans la littérature sur les tests influencée par XP.

Mais trop souvent je vois que les objets simulacres sont mal décrits. En particulier je les vois souvent confondus avec les bouchons - des utilitaires souvent employés dans les environnements de tests. Je comprends cette confusion - je les ai considérés comme similaires pendant un moment, mais des conversations avec les développeurs de simulacres ont fait régulièrement pénétrer un peu de compréhension des simulacres dans mon crâne de tortue.

Cette différence est en fait composée de deux différences distinctes. D'une part il y a une différence dans la vérification des tests : il faut distinguer vérification d'état et vérification de comportement. D'autre part il y a une différence de philosophie dans la manière dont le test et la conception interagissent, ce que je nomme ici par style "classique" et "orienté-simulacre" du développement dirigé par les tests.

(Dans la version précédente de cet essai, j'avais réalisé qu'il y avait une différence, mais je combinais les deux ensemble. Depuis, ma compréhension s'est améliorée et il est temps de mettre à jour cet essai. Si vous n'avez pas lu le précédent essai vous pouvez ignorer mes douleurs croissantes, car j'ai écrit cet essai comme si la précédente version n'existait pas. Mais si vous êtes familier avec la précédente version, vous pourriez trouver utile de noter que j'ai cassé la vieille dichotomie des tests basés sur l'état / tests basés sur l'interaction en deux dichotomies : celle de la vérification état/comportement et celle du style classique/orienté-simulacre. J'ai également ajusté mon vocabulaire pour qu'il corresponde à celui du livre de Gerard Meszaros **xUnit Test Patterns**).

Les tests traditionnels

Je vais commencer par illustrer les deux styles avec un exemple simple (l'exemple est en Java, mais les principes sont valables pour n'importe quel langage orienté-objet). Nous voulons prendre un objet Commande (Order) et le remplir à partir d'un objet Entrepôt (Warehouse). La commande est très simple, avec un seul produit et une quantité. L'entrepôt contient les inventaires de différents produits. Quand nous demandons à une commande de se remplir elle-même à partir d'un entrepôt, il y a deux réponses possibles. S'il y a suffisamment de produit dans l'entrepôt pour satisfaire la commande, la commande est considérée remplie, et dans l'entrepôt la quantité de produit est réduite du montant approprié. S'il n'y a pas suffisamment de produit alors la commande n'est pas remplie, et rien ne se produit au niveau de l'entrepôt.

Ces deux comportements impliquent quelques tests, qui sont des tests JUnit assez conventionnels :

```
public class OrderStateTester extends TestCase {
  private static String TALISKER = "Talisker";
  private static String HIGHLAND_PARK = "Highland Park";
  private Warehouse warehouse = new WarehouseImpl();

  protected void setUp() throws Exception {
    warehouse.add(TALISKER, 50);
    warehouse.add(HIGHLAND_PARK, 25);
  }
  public void testOrderIsFilledIfEnoughInWarehouse() {
    Order order = new Order(TALISKER, 50);
    order.fill(warehouse);
    assertTrue(order.isFilled());
    assertEquals(0, warehouse.getInventory(TALISKER));
  }
  public void testOrderDoesNotRemoveIfNotEnough() {
    Order order = new Order(TALISKER, 51);
    order.fill(warehouse);
    assertFalse(order.isFilled());
    assertEquals(50, warehouse.getInventory(TALISKER));
  }
```

Note du traducteur : cet exemple est disponible en C# **ici**.

Les tests xUnit suivent une séquence typique de quatre phases : initialisation, exécution, vérification, nettoyage. Dans ce cas la phase d'initialisation est faite partiellement dans la méthode setUp (initialiser l'entrepôt) et partiellement dans la méthode de test (initialisation de la commande). L'appel à *order.fill* est la phase d'exécution. C'est là que l'objet est incité à faire la chose que nous voulons tester. Les déclarations assert sont ensuite la phase de vérification, elles contrôlent si la méthode exécutée a fait correctement son travail. Dans ce cas il n'y a pas de phase de nettoyage explicite, car le ramasse-miettes le fait implicitement pour nous.

Durant l'initialisation il y a deux sortes d'objets que nous mettons ensemble. Order est la classe que nous testons, mais pour que *order.fill* fonctionne, nous avons également besoin d'une instance de Warehouse. Dans cette situation Order est l'objet sur lequel nous focalisons le test. Les gens orientés-tests aiment utiliser des termes comme l'"objet en cours de test" ou le "système en cours de test" pour nommer une telle chose. Chacun de ces termes est difficile à dire, mais comme ils sont largement acceptés je me force à les utiliser. Suivant Meszaros j'utilise Système en cours de test, ou encore l'abréviation SCT.

Donc pour ce test j'ai besoin du SCT (*Order*) et d'un collaborateur (*Warehouse*). J'ai besoin de Warehouse pour deux raisons : d'une part pour faire fonctionner le comportement testé (puisque *order.fill* appelle les méthodes de warehouse) et d'autre part pour la vérification (puisqu'un des résultats de order.fill est un changement potentiel dans l'état de Warehouse). Au fur et au mesure que nous allons explorer ce sujet, vous allez voir que nous insisterons

beaucoup sur la distinction entre le SCT et les collaborateurs (dans la version plus ancienne de cet article je parlais du SCT comme l'"objet primaire" et des collaborateurs comme les "objets secondaires").

Ce type de test utilise la **vérification de l'état**, ce qui signifie que nous déterminons si la méthode exécutée a fonctionné correctement en examinant l'état du SCT et de ses collaborateurs après l'exécution de la méthode. Comme nous le verrons, les objets simulacres permettent une approche différente de la vérification.

Les tests avec des objets simulacres

Maintenant je prends exactement le même comportement mais j'utilise des objets simulacres. Pour ce code j'utilise la bibliothèque jMock pour définir les simulacres. jMock est une bibliothèque java pour les objets simulacres. Il y a d'autres bibliothèques pour objets simulacres, mais celle-ci est à jour et est développée par les créateurs de cette technique, donc c'est un bon point de départ.

```
public class OrderInteractionTester extends MockObjectTestCase {
  private static String TALISKER = "Talisker";

  public void testFillingRemovesInventoryIfInStock() {
    //setup - data
    Order order = new Order(TALISKER, 50);
    Mock warehouseMock = new Mock(Warehouse.class);

    //setup - expectations
    warehouseMock.expects(once()).method("hasInventory")
      .with(eq(TALISKER),eq(50))
      .will(returnValue(true));
    warehouseMock.expects(once()).method("remove")
      .with(eq(TALISKER), eq(50))
      .after("hasInventory");

    //exercise
    order.fill((Warehouse) warehouseMock.proxy());

    //verify
    warehouseMock.verify();
    assertTrue(order.isFilled());
  }

  public void testFillingDoesNotRemoveIfNotEnoughInStock() {
    Order order = new Order(TALISKER, 51);
    Mock warehouse = mock(Warehouse.class);

    warehouse.expects(once()).method("hasInventory")
      .withAnyArguments()
      .will(returnValue(false));

    order.fill((Warehouse) warehouse.proxy());

    assertFalse(order.isFilled());
  }
```

Note du traducteur : cet exemple est disponible en C# **ici**.

Concentrez vous d'abord sur le test *testFillingRemovesInventoryIfInStock* , car j'ai pris quelques raccourcis avec l'autre test.

Tout d'abord, la phase d'initialisation est très différente. Pour commencer, elle est divisée en deux parties : les données et les attentes. La partie données initialise les objets qui nous intéressent, et en ce sens elle est similaire à l'initialisation traditionnelle. La différence réside dans les objets qui sont créés. le SCT est le même - une commande. Cependant le collaborateur n'est plus un entrepôt, mais un simulacre d'entrepôt - techniquement une instance de la classe Mock.

La deuxième partie de l'initialisation crée des attentes sur l'objet simulacre. Les attentes indiquent quelles méthodes devraient être appelées sur les simulacres quand le SCT est exécuté.

Une fois que les attentes sont en place, j'exécute le SCT. Après l'exécution je fais alors la vérification, qui a deux aspects. J'utilise des assertions sur le SCT - à peu près comme avant. Cependant je vérifie également les simulacres - en contrôlant qu'ils ont bien été appelés selon leurs attentes.

La différence clé ici est comment nous vérifions que la commande a fait ce qu'elle devait dans son interaction avec l'entrepôt. Avec la vérification d'état, nous faisons cela avec des assertions sur l'état de l'entrepôt. Les simulacres utilisent la **vérification du comportement**, et nous vérifions alors si la commande a fait les bons appels de méthodes sur l'entrepôt. Nous faisons cela pendant l'initialisation en disant au simulacre ce qu'il doit attendre, puis en demandant au simulacre de se vérifier lui-même durant la phase de vérification. Seule la commande est vérifiée à l'aide d'assertions, et si la méthode testée ne change pas l'état de la commande, alors il n'y a même pas d'assertions du tout.

Dans le deuxième test je fais plusieurs choses différemment. Premièrement je crée le simulacre d'une autre manière, en utilisant la méthode *mock* dans MockObjectTestCase au lieu du constructeur. C'est une méthode utilitaire de la bibliothèque jMock, qui me permet d'éviter de faire explicitement la vérification plus tard ; en effet tout simulacre crée avec cette utilitaire est automatiquement vérifié à la fin du test. J'aurais pu faire cela avec le premier test aussi, mais je voulais montrer la vérification d'une manière plus explicite pour bien montrer comment fonctionne le test avec des simulacres.

Deuxièmement, dans le second test, j'ai relâché les contraintes sur l'attente en utilisant *withAnyArguments*. La raison est que le premier test vérifie déjà que le nombre est bien passé à l'entrepôt, aussi le deuxième test n'a pas besoin de répéter cet élément de test. Si la logique de la commande doit être modifiée plus tard, alors un seul test échouera, ce qui facilitera l'effort de migration des tests. J'aurais également pu laisser entièrement de côté *withAnyArguments*, car c'est le fonctionnement par défaut.

Utilisation de EasyMock

Il y a un certain nombre de bibliothèques d'objets simulacres. Je rencontre assez fréquemment EasyMock, à la fois dans ses versions java et .NET. EasyMock permet également la vérification du comportement, mais a plusieurs différences de style avec jMock qui méritent d'être discutées. Voici à nouveau nos tests :

```
public class OrderEasyTester extends TestCase {
  private static String TALISKER = "Talisker";

  private MockControl warehouseControl;
  private Warehouse warehouseMock;

  public void setUp() {
    warehouseControl = MockControl.createControl(Warehouse.class);
    warehouseMock = (Warehouse) warehouseControl.getMock();
  }

  public void testFillingRemovesInventoryIfInStock() {
    //setup - data
    Order order = new Order(TALISKER, 50);

    //setup - expectations
    warehouseMock.hasInventory(TALISKER, 50);
    warehouseControl.setReturnValue(true);
    warehouseMock.remove(TALISKER, 50);
    warehouseControl.replay();

    //exercise
    order.fill(warehouseMock);

    //verify
    warehouseControl.verify();
    assertTrue(order.isFilled());
```

```
    }

    public void testFillingDoesNotRemoveIfNotEnoughInStock() {
      Order order = new Order(TALISKER, 51);

      warehouseMock.hasInventory(TALISKER, 51);
      warehouseControl.setReturnValue(false);
      warehouseControl.replay();

      order.fill((Warehouse) warehouseMock);

      assertFalse(order.isFilled());
      warehouseControl.verify();
    }
  }
```

Note du traducteur : cet exemple est disponible en C# **ici**.

EasyMock utilise la métaphore enregistrer/rejouer pour définir les attentes. Pour chaque objet collaborateur pour lequel vous désirez un simulacre, vous créez un objet de contrôle et un simulacre. Le simulacre implémente l'interface de l'objet collaborateur, tandis que l'objet de contrôle vous donne des fonctionnalités supplémentaires. Pour indiquer une attente, vous appelez la méthode sur le simulacre, avec les arguments que vous attendez. Vous faites ensuite un appel au contrôle si vous voulez une valeur de retour. Une fois que vous avez fini de définir les attentes, vous appelez replay sur le contrôle - à ce stade le simulacre termine l'enregistrement et est prêt à répondre au SCT. Une fois que c'est fait vous appelez verify sur le contrôle.

Il semble que les gens sont souvent troublés à la première vue de la métaphore enregistrer/rejouer, mais qu'ils s'y habituent rapidement. Elle a un avantage par rapport aux contraintes de jMock en ce que vous faites de vrais appels de méthodes sur le simulacre au lieu de spécifier des noms de méthodes dans des chaînes. Ce qui veut dire que vous pouvez utiliser la complétion de code de votre environnement de développement, et que tout remaniement de noms de méthodes mettra automatiquement à jour les tests. L'inconvénient est que vous ne pouvez pas relâcher les contraintes.

Les développeurs de jMock travaillent sur une nouvelle version qui permettra d'utiliser de vrais appels de méthodes.

La différence entre les simulacres et les bouchons

Quand ils ont été introduits pour la première fois, beaucoup de gens ont facilement confondu les objets simulacres avec la notion courante de test avec des bouchons. Depuis ils semblent que les différences sont mieux comprises (et j'espère que la précédente version de cet article y a contribué). Cependant, pour comprendre complètement comment les simulacres sont utilisés, il est important de comprendre à la fois les simulacres et les autres types de doublures de tests ("doublures" ? Ne vous inquiétez pas si c'est un nouveau terme pour vous, attendez quelques paragraphes et tout deviendra clair).

Quand vous testez, vous vous focalisez sur un seul élément du logiciel à la fois - d'où le terme courant de test unitaire. Le problème est que pour faire fonctionner un élément particulier, vous avez souvent besoin d'autres éléments - par exemple dans notre exemple nous avons besoin de Warehouse.

Dans les deux types de tests que j'ai montré ci-dessus, le premier type utilise un vrai objet Warehouse, et le deuxième utilise un simulacre de Warehouse, lequel bien sûr n'est pas un vrai objet Warehouse. Utiliser un simulacre est donc un moyen de ne pas utiliser un vrai Warehouse dans le test, mais d'autres formes de "faux" objets sont également utilisées.

Le vocabulaire pour parler de ces notions devient vite confus - toutes sortes de mots sont utilisés : bouchon, simulacre, substitut, fantôme. Pour cet article je vais suivre le vocabulaire du livre de Gerard Meszaros. Il n'est pas utilisé par tout le monde, mais je pense que c'est un bon vocabulaire, et comme il s'agit de mon article j'ai le privilège de choisir quels mots utiliser.

Meszaros utilise le terme Doublure de Test comme terme générique pour tout objet utilisé à la place d'un vrai objet dans le but de tester. Ce terme correspond au cascadeur qui double un acteur dans un film (un des buts de Meszaros était d'éviter tout nom déjà largement utilisé). Meszaros définit alors quatre types particuliers de doublures:

- Fantômes : des objets que l'on fait circuler, mais qui ne sont jamais réellement utilisés. Habituellement ils servent juste à remplir des listes de paramètres.
- Substituts : des objets qui ont de véritables implémentations qui fonctionnent, mais qui généralement prennent des raccourcis qui les rendent impropre à l'utilisation en production (un bon exemple est une **base de données en mémoire** au lieu d'une vraie base de données).
- Bouchons : ces objets fournissent des réponses prédéfinies à des appels faits durant le test, mais généralement ne répondent à rien d'autre en dehors de ce qui leur a été programmé pour le test. Les bouchons peuvent aussi enregistrer de l'information concernant les appels, par exemple un bouchon de passerelle de courriels peut mémoriser les messages qu'il a "envoyé", ou encore seulement le nombre de messages qu'il a "envoyés".
- Simulacres : les objets dont nous parlons ici: des objets préprogrammés avec des attentes, lesquelles constituent une spécification des appels qu'ils s'attendent à recevoir.

Parmi toutes ces doublures, seuls les simulacres insistent sur la vérification du comportement. Les autres doublures peuvent (et généralement le font) utiliser la vérification d'état. En fait les simulacres se comportement vraiment comme les autres doublures dans la phase d'exécution, car ils ont besoin de faire croire au SCT qu'il parle à ses vrais collaborateurs - mais les simulacres diffèrent dans les phases d'initialisation et de vérification.

Pour explorer un peu plus les doublures de test, nous devons étendre notre exemple. Beaucoup de gens utilisent une doublure uniquement si le vrai objet est peu pratique à manipuler. Ici, disons que nous voulons envoyer un courriel si un Order échoue. Le problème est que nous ne pouvons pas envoyer de vrais courriels à des clients pendant nos tests. Donc à la place nous créons une doublure de notre système de courriels, que nous pouvons contrôler et manipuler.

Nous pouvons alors commencer à voir la différence entre les simulacres et les bouchons. Pour tester ce comportement d'envoi de courriels, nous pourrions écrire un simple bouchon comme ceci :

```
public interface MailService {
  public void send (Message msg);
}

public class MailServiceStub implements MailService {
  private List<Message> messages = new ArrayList<Message>();
  public void send (Message msg) {
    messages.add(msg);
  }
  public int numberSent() {
    return messages.size();
  }
}
```

Nous pouvons alors utiliser la vérification d'état sur le bouchon comme suit :

```
class OrderStateTester...
  public void testOrderSendsMailIfUnfilled() {
    Order order = new Order(TALISKER, 51);
    MailServiceStub mailer = new MailServiceStub();
    order.setMailer(mailer);
    order.fill(warehouse);
    assertEquals(1, mailer.numberSent());
  }
```

Bien sûr c'est un test très simple - il vérifie seulement qu'un message a été envoyé. Nous n'avons pas testé qu'il a été envoyé à la bonne personne, ni qu'il avait le bon contenu, mais il suffira à illustrer mon propos.

En utilisant des simulacres, ce test serait bien différent :

```
class OrderInteractionTester...
  public void testOrderSendsMailIfUnfilled() {
    Order order = new Order(TALISKER, 51);
    Mock warehouse = mock(Warehouse.class);
    Mock mailer = mock(MailService.class);
    order.setMailer((MailService) mailer.proxy());

    mailer.expects(once()).method("send");
    warehouse.expects(once()).method("hasInventory")
      .withAnyArguments()
      .will(returnValue(false));

    order.fill((Warehouse) warehouse.proxy());
  }
}
```

Dans les deux cas j'utilise une doublure de test à la place du vrai service de courriel. La différence est dans le fait que le bouchon utilise la vérification d'état alors que le simulacre utilise la vérification du comportement.

Pour utiliser la vérification d'état sur le bouchon, j'ai besoin de quelques méthodes supplémentaires sur le bouchon pour faciliter la vérification. Par conséquent le bouchon implémente *MailService* mais ajoute des méthodes de test supplémentaires.

Les simulacres utilisent toujours la vérification du comportement, tandis qu'un bouchon peut faire les deux. Meszaros nomme les bouchons qui utilisent la vérification de comportement comme des Espions de Test. La différence réside dans la manière dont la doublure effectue exécution et vérification, et je vous laisse le soin d'explorer cela vous-même.

Les styles de test classique et orienté-simulacre

Maintenant j'en suis au point où je peux explorer la deuxième dichotomie entre le test classique et orienté-simulacre. Le point principal est *quand* utiliser un simulacre (ou une autre doublure).

Le **style classique de TDD** consiste à utiliser de vrais objets si possibles, et une doublure quand le vrai objet est trop compliqué à utiliser. Ainsi un adepte du TDD classique utiliserait le vrai Warehouse et un double pour le service de courriels. Le type exact de doublure ne compte pas tant que cela.

Un **adepte du style orienté-simulacre**, par contre, utilisera toujours un simulacre pour tout objet ayant un comportement intéressant, et donc dans ce cas pour la à la fois Warehouse et le service de courriels.

Bien que les différents outils de création de simulacres aient été conçus avec le style orienté-simulacre en tête, beaucoup d'adeptes du style classique les trouvent utiles pour créer des doubles.

Une conséquence importante du style orienté-simulacre est le **Développement Dirigé par le Comportement** (BDD). Le BDD a été initialement développé par mon collègue Dan North comme une technique pour aider les gens à mieux apprendre le Développement Dirigé par les Tests en insistant sur la technique de conception que représente le TDD. Cela a conduit à renommer les tests en comportements pour mieux explorer comment le TDD aide à réfléchir à ce qu'un objet doit faire. Le BDD suit une approche orientée-simulacre, mais il va plus loin, à la fois dans ses styles de nommage, et dans son désir d'intégrer la conception dans sa technique. Je ne vais pas plus loin ici, car le seul lien avec cet article est que le BDD est une autre variation du TDD qui tend à utiliser les simulacres. Je vous laisse suivre le lien pour plus d'informations.

Comment choisir parmi les différences ?

Dans cet article j'ai expliqué deux différences : vérification de l'état ou du comportement, style classique ou orienté simulacre. Quels sont les arguments à garder l'esprit quand on fait un choix entre ces styles ? Je vais commencer par le choix entre la vérification de l'état ou du comportement.

La première chose à considérer est le contexte. Pensons-nous à une collaboration facile, comme entre Order et Warehouse, ou une compliquée, comme entre Order et le service de courriels ?

Si c'est une collaboration facile alors le choix est simple. Si je suis un adepte du style classique, je n'utilise pas de simulacre, bouchon ou autre sorte de doublure. J'utilise le vrai objet et la vérification d'état. Si je suis un adepte du style orienté-simulacre, j'utilise un simulacre et la vérification du comportement. Pas de décision à prendre du tout.

Si c'est une collaboration compliquée, il n'y a pas de décision si je suis un adepte des simulacres - j'utilise juste des simulacres et la vérification du comportement. Si je suis un adepte du style classique, alors j'ai le choix, mais il n'est pas bien important. Habituellement les adeptes du style classique décident au cas par cas, en utilisant le chemin le plus rapide pour chaque situation.

Donc comme nous le voyons, la décision entre vérification de l'état ou du comportement n'est pas une grosse décision la plupart du temps. La vraie difficulté est de choisir entre TDD classique et orienté-simulacre. Il apparaît que les caractéristiques de la vérification de l'état et du comportement affectent cette discussion, et c'est là que je vais focaliser mon énergie.

Mais avant de le faire, je vais proposer un cas limite. De temps en temps vous rencontrez des choses dont il est vraiment difficile de vérifier l'état, même s'il ne s'agit pas de collaborations compliquées. Un bon exemple de cela est un cache. La particularité d'un cache est que vous ne pouvez pas dire à partir de son état comment le cache a fonctionné - c'est un cas où la vérification du comportement serait un choix sage même pour un adepte pur et dur du style TDD classique. Je suis sûr qu'il y a d'autres exceptions dans les deux directions.

Maintenant que nous approfondissons le choix entre styles classique/orienté-simulacre, il y a des tas de facteurs à considérer, donc je les ai organisés en groupes grossiers :

Le guidage du TDD

Les objets simulacres sont issus de la communauté XP, et l'une des caractéristiques principales de XP est son insistance sur le Développement Dirigé par les Tests - dans lequel la conception d'un système évolue par des itérations pilotées par l'écriture de tests.

Ainsi il n'y a rien de surprenant à ce que les adeptes du style orienté-simulacre parlent de l'effet sur la conception du test avec des simulacres. En particulier ils recommandent un style appelé Développement Dirigé par les Besoins. Avec ce style vous commencez à développer une histoire utilisateur en écrivant votre premier test pour l'extérieur de votre système, en faisant d'un certain objet interface votre SCT. En réfléchissant aux attentes sur les collaborateurs, vous explorez les interactions entre le SCT et ses voisins - et donc vous concevez effectivement l'interface externe du SCT.

Une fois que vous avez votre premier test qui tourne, les attentes sur les simulacres fournissent les spécifications pour la prochaine étape, et un point de départ pour les tests. Vous transformez chaque attente en un test sur un collaborateur et répétez le processus en progressant dans le système un SCT à la fois. Ce style est également nommé "extérieur-vers-intérieur", ce qui en est une très bonne description. Il fonctionne bien avec les systèmes organisés en couches. Vous commencez d'abord en programmant l'interface utilisateur en utilisant des simulacres des couches sous-jacentes. Ensuite vous écrivez vos tests pour la couche en-dessous, en parcourant graduellement le système

une couche à la fois. C'est une approche très structurée et très contrôlée, dont beaucoup de gens pensent qu'elle est utile pour guider les novices dans la programmation orientée-objet et le TDD.

Le TDD classique ne fournit pas tout à fait le même guidage. Vous pouvez faire une approche progressive similaire, en utilisant des méthodes bouchons à la place de simulacres. Pour faire cela, chaque fois que vous avez besoin de quelque chose d'un collaborateur, vous codez simplement en dur la réponse nécessaire pour faire marcher le SCT. Une fois que vous avez la barre verte vous remplacez la réponse en dur par le code approprié.

Mais le TDD classique peut faire d'autres choses également. Un style courant est "milieu-vers-extérieur". Dans ce style vous prenez une fonctionnalité et décidez ce dont vous avez besoin dans le domaine pour que cette fonctionnalité marche. Vous faites faire aux objets du domaine ce dont vous avez besoin, et une fois qu'ils marchent vous établissez l'interface utilisateur au-dessus. En faisant cela vous pouvez très bien ne rien avoir à doubler du tout. Beaucoup de gens aiment cette approche car elle concentre l'attention sur le modèle du domaine d'abord, ce qui empêche la logique du domaine de contaminer l'interface utilisateur.

Je voudrais souligner que les adeptes des styles orienté-simulacre et classique font cela une seule histoire utilisateur à la fois. Il y a une école de pensée qui construit les applications couche par couche, ne commençant pas une nouvelle couche tant que celle en cours n'est pas terminée. Au contraire les adeptes du style classique et des simulacres tendent à avoir une approche agile et préfèrent des itérations de petite taille. Par conséquent ils travaillent fonctionnalité par fonctionnalité plutôt que couche par couche.

L'initialisation des classes de test

Avec le TDD classique vous devez créer non pas seulement le SCT mais aussi tous les collaborateurs dont le SCT à besoin pour répondre au test. Bien que l'exemple ci-dessus n'ait que peu d'objets, les tests réels impliquent souvent un grand nombre de collaborateurs. Habituellement ces objets sont créés et supprimés lors de chaque exécution des tests.

Les tests avec simulacres, cependant, ont seulement besoin de créer le SCT et des simulacres pour ses collaborateurs immédiats. Ceci peut éviter un peu du travail impliqué dans la construction de classes de test complexes (au moins en théorie. J'ai entendu des histoires d'initialisations de simulacres pas mal complexes, mais cela était peu être dû à une mauvaise utilisation des outils).

En pratique les testeurs classiques tendent à réutiliser autant que possible les initialisations de test complexes. La manière la plus simple est de mettre le code d'initialisation dans une méthode setup xUnit. Des initialisations plus compliquées peuvent avoir besoin d'être utilisées par plusieurs classes de test, alors dans ce cas vous créez des classes spéciales pour générer les initialisations. Je les appelle habituellement des **Mères d'objets**, en me basant sur une convention de nommage utilisée dans un projet XP précoce chez ThoughtWorks. Utiliser des mères est essentiel dans le test classique de grande envergure, mais les mères représentent du code supplémentaire qui a besoin d'être maintenu, et tout changement au niveau des mères peut avoir des répercussions en chaîne à travers les tests. Il peut aussi y avoir une pénalité de performance à l'initialisation - bien que je n'ai pas entendu dire que ce soit un problème sérieux si cela est fait correctement. La création de la plupart des objets d'initialisation est bon marché, et quand ce n'est pas le cas ils sont habituellement doublés.

En conséquence j'ai entendu chaque style accuser l'autre de représenter trop de travail. Les partisans des simulacres disent que créer les initialisations est un gros effort, mais les classiques disent qu'elles sont réutilisables alors qu'il faut créer des simulacres pour chaque test.

L'isolation des tests

Si vous introduisez un bug dans un système avec du test basé sur des simulacres, généralement il fera échouer uniquement les tests dont le SCT contient le bug. Avec l'approche classique, cependant, n'importe quel test d'objet client peut aussi échouer, ce qui conduit à des échecs aux endroits où l'objet fautif est utilisé comme collaborateur dans le test d'un autre objet. Par conséquent, un échec dans un objet très utilisé cause une cascade d'échecs de test à travers le système.

Les testeurs utilisant des simulacres considèrent cela comme un problème majeur; il conduit à beaucoup de débogage pour trouver la cause racine de l'erreur et la corriger. Cependant les classiques n'expriment pas cela comme une source de problèmes. Habituellement le coupable est assez facile à identifier en examinant quels tests échouent et les développeurs peuvent alors dire quels échecs dérivent de la cause racine. De plus si vous testez régulièrement (comme vous le devriez) alors vous savez que l'échec a été causé par ce que vous avez édité en dernier, donc il n'est pas difficile de trouver l'erreur.

La granularité des tests peut être un facteur significatif ici. Puisque les tests classiques mettent en jeu de nombreux objets réels, vous trouvez souvent un test particulier qui est le test primaire pour un groupe d'objets, plutôt que pour un seul objet. Si ce groupe comprend beaucoup d'objets, alors trouver la vraie cause d'un bug peut être beaucoup plus difficile. Ce qui arrive ici c'est ce que les tests ont une granularité trop grossière.

Il est probable que les tests avec des simulacres sont moins enclins à souffrir de ce problème, car la convention est de faire des simulacres pour tous les objets au-delà de l'objet primaire, ce qui rend clair que des tests de granularité plus fine sont nécessaires pour les collaborateurs. Ceci étant dit, il est aussi vrai qu'utiliser des tests à granularité trop grossière n'est pas nécessairement un échec du test classique en tant que technique, c'est plutôt dû à une mauvaise application du test classique. Une bonne règle empirique est de s'assurer d'isoler des tests de fine granularité pour chaque classe. Alors que des groupes sont parfois raisonnables, ils devraient être limités à un tout petit nombre d'objets - pas plus qu'une douzaine. De plus, si vous vous trouvez face à un problème de débogage dû à des tests de granularité trop grossière, vous devriez déboguer dans une optique TDD, en créant des tests de plus fine granularité au fur et à mesure que vous avancez.

En l'essence les tests classiques xunit ne sont pas juste des tests unitaires, mais aussi de mini tests d'intégration. Par conséquent beaucoup de gens aiment le fait que des tests sur des clients peuvent attraper des erreurs que les tests principaux pour un objet auraient ratées, en sondant particulièrement les zones où les classes interagissent. Les tests avec les simulacres perdent cette qualité. De plus vous courrez aussi le risque que les attentes sur les simulacres soient incorrectes, ce qui résulte en des tests unitaires qui passent au vert mais masquent des erreurs inhérentes.

A ce stade je dois souligner que quelque soit le style de test que vous utilisez, vous devez le combiner avec des tests d'acceptance de granularité plus grossière, tests qui opèrent à travers tout l'ensemble du système. J'ai souvent croisé des projets qui étaient en retard dans l'utilisation de tests d'acceptance, et qui l'ont regretté.

Le couplage des tests avec les implémentations

Quand vous écrivez un test avec simulacres, vous testez les appels du SCT vers l'extérieur pour vous assurer qu'il parle correctement à ses fournisseurs. Un test classique s'intéresse uniquement à l'état final - pas à la manière dont cet état a été obtenu. Les tests avec simulacres sont ainsi plus fortement couplés à l'implémentation d'une méthode. Changer la nature des appels aux collaborateurs cassera généralement un test avec simulacre.

Ce couplage entraîne plusieurs préoccupations. La plus importante est l'effet sur le TDD. Dans le cas des tests avec simulacres, écrire le test vous fait réfléchir à l'implémentation du comportement - et en effet les testeurs avec simulacres voient cela comme un avantage. Les classiques, cependant, pensent qu'il est important de réfléchir seulement à ce qui arrive de l'interface externe, et de laisser de côté toute considération d'implémentation jusqu'à ce que vous ayez fini d'écrire le test.

Le couplage avec l'implémentation interfère également avec le remaniement, puisque les changements d'implémentation risquent beaucoup plus de casser les tests que dans le cas du test classique.

Cela peut être aggravé par la nature même des boites à outils de simulacres. Souvent les outils de simulacres spécifient des appels de méthodes et des appariements de paramètres très spécifiques, même quand ils ne sont pas pertinents pour le test en question. Un des buts de la boite à outils jMock est d'être plus flexible dans sa spécification des attentes, pour autoriser le relâchement des attentes dans les zones où elles n'ont pas d'importance, au prix de l'utilisation de chaînes qui rendent les remaniements plus délicats.

Le style de conception

Pour moi, l'un des aspects les plus fascinants de ces styles de test est la manière dont ils influencent les décisions de conception. Comme j'ai parlé avec les deux types de testeurs, je suis devenu conscient de quelques différences entre les conceptions encouragées par les styles, mais je suis certain que je ne fais qu'égratigner la surface de ce sujet.

J'ai déjà mentionné une différence dans la manière d'aborder les couches. Les tests avec simulacres supportent une approche "extérieur-vers-intérieur" tandis que les développeurs qui préfèrent travailler à partir du modèle du domaine tendent à préférer le test classique.

A un niveau moins élevé, j'ai noté que les testeurs orientés-simulacres tendent à s'éloigner des méthodes qui retournent des valeurs, pour favoriser les méthodes qui agissent sur un objet collecteur. Prenez l'exemple du comportement consistant à rassembler de l'information d'un groupe d'objets pour créer un rapport sous forme de chaîne de caractères. Une manière courante de procéder est d'avoir une méthode de rapport qui appelle sur les différents objets des méthodes retournant des chaînes, et qui assemble la chaîne résultat dans une variable temporaire. Un testeur orienté-simulacre passera probablement un tampon de caractères aux différents objets et leur fera ajouter les différentes chaînes à ce tampon - traitant ainsi le tampon comme un paramètre collecteur.

Les testeurs avec simulacres parlent également plus d'éviter les désastres en chaînes - les chaînes de méthodes du type *getThis().getThat().getTheOther()*. Éviter les chaînes de méthodes est également connu sous le nom de la Loi de Demeter. Alors que les chaînes de méthodes sont une mauvaise odeur, le problème opposé des objets intermédiaires boursouflés de méthodes de transfert est également une mauvaise odeur. (J'ai toujours senti que je serais plus confortable avec la Loi de Demeter si elle était appelée la Suggestion de Demeter).

L'une des choses les plus difficiles à comprendre dans la conception orientée-objet est le principe **"Fais au lieu de demander"** qui vous encourage à dire à un objet de faire quelque chose, au lieu de lui extraire des données pour faire la chose en question dans du code client. Les testeurs orienté-simulacres disent que le test avec simulacres favorise ce principe et évite les confettis de code "get..." qui se répand dans beaucoup trop de code ces temps-ci. Les classiques avancent qu'il y a beaucoup d'autres manières de faire cela.

Un problème reconnu avec la vérification d'état est qu'elle peut conduire à la création de méthodes requêtes servant seulement à supporter la vérification. Il n'est jamais confortable d'ajouter des méthodes à l'API d'un objet dans le seul but de tester; utiliser la vérification du comportement évite ce problème. Le contre-argument est que de telles modifications sont souvent mineures en pratique.

Les testeurs orientés-simulacres favorisent les **interfaces de rôles** et assurent qu'utiliser ce style de test encourage la création de plus d'interfaces de rôles, puisque chaque collaboration a son propre simulacre et est donc plus sujette à devenir une interface de rôle. Ainsi dans mon exemple ci-dessus concernant la génération d'un rapport en utilisant un tampon de caractères, un testeur orienté-simulacre serait plus enclin à inventer un rôle particulier qui aurait du sens dans ce domaine, et qui pourrait être implémenté par un tampon de chaînes.

Il est important de se rappeler que cette différence de style de conception est un élément de motivation clé pour la plupart des testeurs orientés-simulacres. Les origines du TDD étaient un désir d'obtenir de solides tests

automatisés de régression qui supporteraient une conception évolutionnaire. Sur le chemin ses adeptes ont découvert qu'écrire des tests d'abord représentait une amélioration significative du processus de conception. Les adeptes du style orienté-simulacre ont une forte idée de quelle conception est une bonne conception, et ont développé des bibliothèques de simulacres principalement pour aider les gens à développer ce style de conception.

Faut-il être un testeur classique ou orienté-simulacres ?

Je trouve qu'il est difficile de répondre avec certitude. Personnellement j'ai toujours été un adepte du bon vieux TDD classique et jusque-là je ne vois pas de raison de changer. Je ne vois aucun bénéfice irréfutable pour le TDD avec simulacres, et je suis préoccupé par les conséquences de coupler les tests avec l'implémentation.

Ceci me frappe particulièrement quand j'observe un programmeur orienté-simulacre. J'aime vraiment le fait que pendant que vous écrivez le test, vous vous concentrez sur le résultat du comportement, pas sur comment il est fait. Un adepte des simulacres réfléchit constamment à l'implémentation du SCT pour pouvoir écrire les attentes. Cela me paraît vraiment anti-naturel.

Je souffre également de l'inconvénient de ne pas essayer le TDD avec simulacres sur autre choses que des applications jouets. Comme je l'ai appris du TDD lui-même, il est souvent difficile de juger une technique sans l'essayer sérieusement. Je connais beaucoup de bons développeurs qui sont des adeptes convaincus et très heureux des simulacres. Donc bien que je sois toujours un classique convaincu, j'ai préféré présenter les deux arguments aussi équitablement que possible de telle sorte que vous puissiez vous faire votre propre idée.

Donc si le test avec simulacres vous paraît attractif, je vous suggère de l'essayer. Cela en vaut particulièrement la peine si vous avez des problèmes dans certaines zones que le test avec simulacre est conçu pour améliorer. Je vois deux zones principales ici. La première si vous passez beaucoup de temps à déboguer quand des tests échouent, parce qu'ils n'échouent pas proprement et ne vous disent pas où est le problème. La deuxième si vos objets ne contiennent pas suffisamment de comportements ; le test avec simulacres peut encourager l'équipe de développement à créer des objets plus riches en comportements.

Considérations finales

Au fur et à mesure que grandit l'intérêt pour les outils xUnit et le Développement Dirigé par les Tests, de plus en plus de gens croisent les simulacres. Le plus souvent, ils apprennent une partie des outils pour simulacres, sans comprendre complètement la division classique/orienté-simulacre sur lesquels ils reposent. Quelque soit votre côté dans cette division, je pense qu'il est utile de comprendre cette différence de vue. Alors que vous n'avez pas besoin d'être orienté-simulacres pour trouver pratiques les outils de simulacres, il est utile de comprendre le raisonnement qui guide la plupart des décisions de conception du logiciel.

Le but de cet article était, et est toujours, de souligner ces différences et d'exposer les compromis entre elles. Il y a plus dans la réflexion orientée-simulacres que ce que j'ai eu le temps d'approfondir, en particulier ses conséquences sur le style de conception. J'espère que dans les toutes prochaines années nous verrons plus de choses écrites sur cela, et que cela approfondira notre compréhension des conséquences fascinantes de l'écriture de tests avant le code.

Lectures complémentaires

Pour une revue complète de la pratique du test unitaire avec xUnit, gardez un #il sur le livre Gerard Meszaros qui va sortir (avertissement : il est dans mes séries). Il maintient aussi un **site web** avec les patrons du livre.

Pour en savoir plus sur le TDD, il faut commencer par le **livre de Kent**.

Pour en savoir plus sur le style de test orienté-simulacre, le premier endroit à visiter est le site **mockobjects.com** où Steve Freeman et Nat Price recommandent le point de vue orienté-simulacre avec des papiers et un bon blog. Lisez en particulier l'excellent papier **OOPSLA**. Pour plus d'informations sur le Développement Dirigé par le Comportement, une variante du TDD très orientée-simulacre, commencez par l'**introduction de Dan North**.

Vous pouvez aussi en savoir plus sur ces techniques en allant voir les sites web des outils **jMock**, **nMock**, **EasyMock**, **EasyMock.net** (il y a d'autres outils disponibles, ne considérez pas cette liste comme complète).

Le **papier original** sur les objets simulacres a été présenté à XP2000, mais il est un peu dépassé maintenant.