6 Les tests et les langages objets

- Introduction
- Le test objet
- Tests structurels
- Test d'interaction

Les langages objets

Caractéristiques des langages de programmation OO

- Liaison dynamique/polymorphisme/couplage faible : le type réel d'un objet n'est pas toujours connu à la compilation, principe des méthodes virtuelles. Lors d'un appel de méthode virtuelle, le comportement de l'objet appelé dépend de son type réel (et de sa hiérarchie de types en cas d'héritage)
- Encapsulation : l'état interne d'un objet n'est pas directement observable, méthodes publiques et attributs privés.
- Héritage : la relation d'héritage n'est pas toujours une relation de sous-typage (sous entendu : comportemental).

Les langages objets

Objet

Objet = état + message (state + behavior) = machine à états

- transitions : méthodes appelées
- états : ensemble des valeurs possibles des attributs internes

État

État observable uniquement au travers de méthodes :

- soit il existe des méthodes explicites (getXXX) ;
- soit observation indirecte par le comportement.

Les langages objets

Héritage et sous-typage

Notion de contrat (4 niveaux) :

- basique : syntaxique (signature de méthodes), système de types
- dynamique : comportement des objets d'une classe, contrats comportementaux, JML, pre- et post-conditions
- synchronisation : synchronisation entre messages, niveau global
- QoS

Les langages objets

- sous-typage des langages à classe : premier niveau
- sous-typage comportemental : deuxième niveau

La réutilisation des tests d'une super-classe implique qu'il existe une relation de sous-typage comportemental. Le test permet de « vérifier » cette relation pour une hiérarchie de classes.

Les langages objets

Héritage des classes de test

On peut réutiliser les tests d'une super-classe uniquement pour les méthodes qui n'affaiblissent pas la pré-condition.

Héritage des classes de test

Comme lors d'un héritage, les invariants et les post-conditions ne peuvent être que renforcés, les tests de la super-classe sont réutilisables.

Exemple d'héritage

Spécification

Soit une classe B qui dérive une classe A. Ces deux classes n'ont qu'une seule méthode foo(**int** x).

Exemple

```
public class A {
    /**
     * Pre-condition : x >= 0
     */
    protected int foo(int x) { ... }
}

public class B extends A {
    /**
     * Pre-condition : x >= -1
     */
    @Override
    protected int foo(int x) { ... }
}
```

Exemple d'héritage

Problème

Un test de B qui va ré-utiliser le test de A pour la pré-condition a toutes les chances d'échouer puisqu'on attend la levée d'une exception lorsque $x = -1$ dans A et pas pour dans B.

Exemple d'héritage

Exemple

```
public class ATest {
    public A instance;

    @Before
    public void setUp() {
        instance = createInstance();
    }

    protected A createInstance() {
        return new A();
    }

    @Test(expected = IllegalArgumentException.class)
    public void preFoo() {
        instance.foo(-1);
    }
}
```

Exemple d'héritage

Exemple

```
public class BTest extends ATest {
    @Override
    protected A createInstance() {
        return new B();
    }
}
```

Exemple d'héritage

Morale

Il faut redéfinir le test de la méthode qui a changé sa pré-condition ou tout simplement ne pas faire hériter les classes de test.

Exemple

```
public class BTest extends ATest {
    @Override
    protected A createInstance() {
        return new B();
    }

    @Test(expected = IllegalArgumentException.class)
    @Override
    public void preFoo() {
        instance.foo(-2);
    }
}
```

6 Les tests et les langages objets

- Introduction
- Le test objet
- Tests structurels
- Test d'interaction

Spécificités du test objet

Caractéristiques des langages objets

- polymorphisme : le concepteur de test doit s'assurer que le code fonctionne correctement quel que soit le type dynamique des objets (pas possible en général, nécessite coopération du concepteur, règles de bonne conduite)
- encapsulation : comment s'assurer que l'état réel de l'objet est bien celui attendu ? Comment positionner l'objet dans un état qu'on souhaite tester (il faudra parfois rajouter des méthodes dans l'interface d'une classe juste pour permettre son test)

Spécificités du test objet

Caractéristiques du processus de développement

- utilisation intensive de modèles objet d'analyse (modèle des exigences ou besoins) et conception (spécification) projetés vers du code
- possibilités de tester ces modèles (donc test au plus tôt) et de raffiner les tests pour obtenir des tests de l'implantation
- avantages : les testeurs peuvent vérifier au plus tôt leur bonne compréhension des besoins/spécifications

Spécificités du test objet

Cas particulier

- test de modèles ;
- test de classe (remplace le test unitaire)
- test d'interaction (remplace le test d'intégration)

Test d'une classe

Le test d'une classe

On a déjà vu lors des tests fonctionnels que pour tester une classe il faut pouvoir tester l'état de ses instances au travers d'inspectrices.

La classe doit être testable

- il faut pouvoir observer ses instances : accès à l'état.
- il faut pouvoir contrôler ses instances : leur état ne doit dépendre que des méthodes qui sont invoquées sur celles-ci.

Test d'une classe

La classe doit être testable unitairement en isolation

- Le résultat du test ne doit pas dépendre du comportement d'autres classes
- Toutes les dépendances entre classes doivent donc être explicites
- Exemple de dépendance non explicite : attribut ne possédant pas d'accesseur dont la création est cachée dans le constructeur

Test d'une classe

Remarque

Autrement dit, on peut tester l'agrégation de classes mais PAS la composition.

Exemple

Dépendance masquée

```
public class Foo {
    public int foo();
}

public class Bar {
    private Foo foo = new Foo();
    private int i;

    public voir bar() {
        i = foo.foo();
    }
}
```

Exemple

Classe non contrôlable

```
public class Toto {

    public boolean etat() {
        switch(new Date().getTime() % 3) {
            case 0: case 1:
                return true ;
            default:
                throw new RuntimeException("bad");
        }
        return false;
    }
}
```

Exemple

Dépendance implicite

```
public class Toto {
    public static final String SAVEFILE = "toto.txt";

    public boolean save() {
        File f = new File(SAVEFILE);
        FileOutputStream fos = new FileOutputStream(f);
        // save state to fos
    }
}
```

Test d'une classe

Faciliter le test

- Faire dépendre autant que possible la classe d'abstractions (par ex. les interfaces java)
- Permet de simuler ces interfaces en les implantant par des bouchons

Problème des tests objets

- Problème des classes non testables unitairement car utilisant des objets créés en interne (xxx = new XXX()). Il faut remplacer les objets par des références d'interfaces et définir des accesseurs/modificateurs ou pouvoir simuler l'objet (mock)
- Problème du code non couvrable par un cas de test
- Problème du test de code utilisant des ressources du système : FS, réseau, hardware, code natif...

Solutions ?

- Définition de tests systèmes
- Définition de tests statistiques à partir d'un profil opérationnel

6 Les tests et les langages objets

- Introduction
- Le test objet
- Tests structurels
- Test d'interaction

Test structurel

Le test MM

Le test MM (méthode, message) consiste à tester chacun des appels de méthode : si une méthode appelle une autre plusieurs fois, chaque appel devra être testé séparément.

Couverture de paire de fonctions

La couverture des paires de fonctions implique de tester tous les enchaînements de deux méthodes possibles. On utilise généralement un diagramme de machine à état ou une expression régulière.

6 Les tests et les langages objets

- Introduction
- Le test objet
- Tests structurels
- Test d'interaction

Test d'interaction

Définition

- test unitaire : test d'une classe en isolation, indépendamment de toute autre classe ;
- test d'intégration : tester un ensemble de classes qui collaborent ;
- test d'interaction : autre nom pour test d'intégration, tester un ensemble de classes qui interagissent entre elles.

Test d'interaction

Le problème

- Test unitaire tel qu'on l'a vu jusqu'à présent : basé sur l'examen de l'état de l'objet (utilisation d'Assert) ;
- stimuler puis observer le résultat sur l'état ;
- et si pas d'état ou état pas modifié ?
- parfois on a juste envie de tester qu'une méthode effectue certains bons appels sur les instances avec lequelles elle collabore ;
- c'est alors un cas particulier de test unitaire à base de « mocks ».

Test d'interaction

Comment tester ?

En écrivant des simulacres à la main : fastidieux !
Il existe des outils qui créent automatiquement des mocks.

Outils

- mockobjects
- DynaMock, EasyMock (successeurs)
- Mockito, jMock : plus récents, tirent parti du retour d'expérience des autres
- VirtualMock (suspendu)
- jBehave : framework plus général que les autres, orienté Behaviour Driven Development

Les doublures

Principe

Principe général de la définition de doublures : objets créés uniquement pour les besoins du test en remplacement d'objets réels.

- Fantôme (Dummy) : objet utilisé uniquement pour son existence, sans qu'aucune interaction avec le CAT ne soit attendue.
- Bouchon (Stub) : fournit des réponses préprogrammés aux appels réalisés par le CAT. Utilisé pour produire des entrées nécessaires au comportement du CAT, définit des points de contrôles
- Espion (Spy) : même principe que le bouchon, mais en plus contrôle les sorties du CAT pour vérification ultérieure. Les vérifications sont faites par examen de l'état de l'espion (attributs spécifiques)

Les doublures

Principe

- Simulacre (Mock) : même principe que l'espion, mais le simulacre est programmé pour vérifier les outputs/produire les inputs au cours du test. Le simulacre implante un scénario d'interactions et vérifie le comportement du CAT par rapport à ce scénario.
- Substitut (Fake) : objet/système remplacant un objet/système utilisé en production pour simplifier le test (ex : utiliser un SGBD résidant en mémoire à la place du vrai SGBD). Le substitut offre tout ou partie des fonctionnalités du substitué mais souvent en mode dégradé.

Les doublures

Différences entre bouchons et simulacres

- Différence d'intention : les bouchons sont utilisés quand on ne peut pas utiliser un vrai objet pour des raisons techniques (effets de bord difficilement testables), les simulacres sont utilisés systématiquement pour tester le comportement du CAT
- différence d'implantation : les bouchons ont le plus souvent une implantation ad hoc, les simulacres sont fournis par un framework

Conséquences

Couplage faible

L'utilisation de ces techniques de test, en particulier en lien avec le TDD a des conséquences importantes sur la conception des classes et des systèmes :

- impose un couplage faible entre les différents objets du système : les objets ne dépendent pas directement des implantations mais d'abstractions (classe abstraites, interfaces) ;
- diminue la granularité des objets : plus les classes sont petites et découpées, plus le test sera facile et rapide ;
- induit l'injection de dépendances pour gérer facilement la connectique entre les objets.

Les objets forment un réseau dont chaque connexion est modifiable (pas nécessairement dynamiquement, mais suffisamment pour permettre d'être testée).

Comportement explicite

Pour écrire correctement les attentes et le résultat de l'invocation d'un mock, il faut connaître avec suffisamment de précision le comportement spécifié de l'objet :

- la notion de contrat devient nécessaire : quels sont les attentes de l'objet, quel est son comportement
- pas nécessairement formalisé, mais au moins doit être explicité.

1 Principes généraux

2 Les tests statiques

3 Introduction aux tests dynamiques

4 Les tests fonctionnels

5 Les tests structurels

6 Les tests et les langages objets

7 Le TDD

Le cycle en V

Développement en cascade

Le cycle de développement en cascade a été la cause de nombreux échecs.

- Les tests apparaissent trop tard ;
- alors qu'il faut détecter les bugs au plus tôt ;
- on a tendance à sauter les tests pour tenir les délais.

Le cycle en V

La cascade : un effet tunnel

On se met d'accord avec le client à l'entrée du tunnel ... et quand on ressort les besoins du client ont changés !

Le cycle en V

La cascade : un effet tunnel

On se met d'accord avec le client à l'entrée du tunnel ... et quand on ressort les besoins du client ont changés !

Grosse phase de spécification

On peut découvrir au codage un problème qui remet en cause le design.

La validation

La seule manière de valider une idée c'est de la faire tester par le client ... tant qu'on en est à la spécification et au modèle on n'a rien de tangible à montrer au client.

Méthodes actuelles de développement

De nos jours

Les techniques modernes de développement utilisent de petites itérations qui peuvent être vues chacunes comme un mini-cycle en V.

Cycles de développement

- itératifs : on enchaine les étapes sur des cycles courts ;
- incrémentaux : fonctionnalité par fonctionnalité.

Méthodes actuelles de développement

Avantages

- Permet une mise au point plus fréquente avec le client ;
- on avance à petit pas : si on rate un pas, il y a moins de dégâts.

Le test dans tout ça

Les méthodes agiles comme eXtreme programming ou Test Driven Development donnent un rôle primordial au test.

Le TDD

Le « Test Driven Development »

- Add a test
- Run all tests and see the new one fail
- Write some code
- Run the automated tests and see them succeed
- Refactor code
- Repeat

Le TDD

Avantages

- On écrit plus de tests et donc on est plus productif ! (étude de 2005)
- On utilise moins (plus) le « débogage » (on revient plutôt à la version précédente)
- Approche complémentaire à la conception par contrats
- Écriture du code par petites étapes, facilitant la concentration du développeur
- Couverture maximale du code par les jeux de test
- Code généralement très modulaire, flexible et extensible
- Impossible de fournir du code non testé.

Le TDD

Pourquoi écrire un seul test à la fois ?

- optique du « petit pas », on progresse comportement par comportement ;
- on ne passe au pas suivant que lorsque le précédent a été validé ;
- on progresse par cycles « je réfléchis un peu ; je code, compile et teste un peu et je recommence ».

Pourquoi ne pas écrire tout le code d'un coup ?

- On n'écrit que le code qui a besoin d'être validé par un test ;
- sinon, emporté par son élan, on risque d'ajouter des fonctionnalités inutiles ;
- exemple : on n'ajoute un accesseur que si on en a besoin pour le test.

Le TDD

Limites

Le TDD est difficilement utilisable sur des systèmes ayant des entrées/sorties complexes ne permettant pas le test de composants isolés comme :

- Les IHM
- Les SGBD

Le BDD

Behaviour Driven Development

Technique de développement inspirée du TDD mais basée sur le test du comportement des objets :

- concentre les tests sur la construction d'un domaine (Domain Driven Design) : un ensemble d'objets représentants un problème dans le système et leurs interactions.
- utilise la technique des simulacres intensivement : chaque objet est testé dans un contexte nécessaire à son exécution qui est simulé
- définit de manière pragmatique une implantation de contrats

Outils

Java

- jBehave
- jDave

JBehave

Deux frameworks pour le prix d'un :

- les Behaviours : framework de type Mock Objects, en un peu plus simple et abstrait
- les Stories : framework pour la définition de tests d'acceptation

Test d'intégration

- Teste un ensemble d'unités entre elles pour vérifier leur bonne intégration.
- Teste une structure d'objets (CAT) particulière correspondant à une ou plusieurs situations en production.
- Teste des configurations probables d'interactions entre objets.

Test d'intégration

Hypothèse

- Le test unitaire n'est pas suffisant ni exhaustif, certaines interactions peuvent ne pas avoir été correctement testées, en particulier lors de séquences d'appels à un même objet.
- Axiome de composition de weyuker : le fait que A et B soient « corrects » n'implique pas que la composition de A et B soit correcte.

Problème

État des objets partagés entre plusieurs instances : les objets doivent coopérer entre eux pour respecter les spécifications des objets qu'ils partagent (ou ne partager que des objets sans états ni effets de bord).

Test d'intégration

Principe

Le test d'intégration doit construire des cas de test simulant les interactions entre les objets et validant leur bonne coopération.

Objectif

Identifier les conflits possibles dans l'utilisation des objets (et du système).