IFT2255 - Génie logiciel

Tests

Bruno Dufour
dufour@iro.umontreal.ca

Activité de test

L'importance des tests

- L'erreur est humaine, presque tous les programmes contiennent des erreurs
- Les erreurs sont difficiles à identifier:
  - Grande complexité des logiciels
  - Invisibilité du système développé
  - Pas de principe de continuité
- L'activité de test est essentielle au développement de logiciels de qualité

Définitions

- Erreur (error): commise par un développeur
  - Erreur de programmation
  - Erreur de logique
- Faute ou défault (fault, defect): état interne invalide d'un logiciel après l'activation d'une erreur
- Défaillance (failure): manifestation externe d'une faute
  - Observable (ex: le programme dévie de sa spécifications)

Erreurs, Fautes & Défaillances

Développement de logiciels fiables

- Prévention des fautes
  - Méthodes formelles (IFT3911)
  - Réutilisation de code
  - Méthode de développement rigoureuses
- Prévision des fautes
  - Métriques de qualité (IFT3913 – H. Sahraoui)
  - Méthodes statistiques
- Identification et correction des fautes
  - Vérification
  - Validation
- Tolérance aux fautes

Tests

- Test: exécution d'un programme dans le but de découvrir des erreurs [Myers 1979], non pas « … dans le but de démontrer l'absence d'erreurs »
  - Démontrer l'absence est très difficile ou impossible même pour de petits programmes
  - « Si on tente de démontrer l'absence d'erreurs, on n'en découvre que très peu. Si on tente de démontrer la présence d'erreurs, on en découvre la grande majorité. » [Myers, 1979]
  - Similairement, « … dans le but de démontrer que le programme fait ce qui est demandé » n'est pas suffisant : un programme peut contenir une erreur s'il fait autre chose en plus de ce qui est demandé, par exemple.

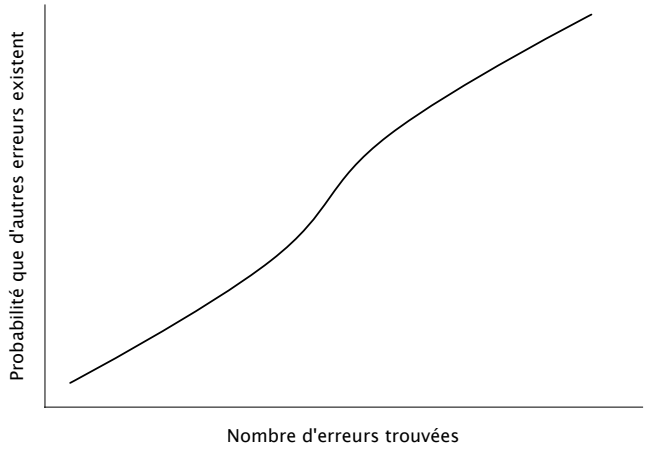
L'importance des tests

- On peut généralement faire l'hypothèse qu'un programme contient des erreurs
- Les tests peuvent être utilisés durant toutes les étapes du développement
  - Avant de débuter l'implémentation (TDD)
  - Durant le développement
  - Après que le logiciel soit complété
- Les tests représentent une partie importante du développement
  - Jusqu'à 33% du budget
  - Jusqu'à 27% du temps de développement

L'art du test

- Bien tester est difficile
  - Tester est un processus destructif
  - Tester une activité très intellectuelle et requiert beaucoup de créativité
- Faire **échouer** le programme est une **réussite** lorsqu'on développe des tests
  - Éventuellement, les tests permettent d'atteindre un niveau de qualité acceptable

Une erreur n'est (presque) jamais seule



Principes de base

- Ne jamais planifier un effort de test sous l'hypothèse tacite qu'aucune erreur ne sera trouvée.
- Un élément nécessaire d'un test est une définition de la sortie ou du résultat
- Un programmeur doit éviter de tester son propre programme
- Les tests doivent être écrits pour les conditions d'entrée qui sont invalides et inattendues, ainsi que pour celles qui sont valides et attendues.
- L'examen d'un programme pour déterminer s'il ne fait pas ce qu'il est censé faire n'est que la moitié de la bataille, l'autre moitié, consiste à déterminer si le programme fait ce qu'il n'est pas censé faire.
- Éviter les tests jetables à moins que le programme soit lui-même vraiment un programme jetable.

L'activité de test

- 2 catégories d'activités :
  - Tests humains : basés sur la lecture du code par un être humain
    - Ex: revues, inspections
  - Tests informatisés : basés sur l'exécution d'un programme par une machine
    - Différents niveaux de granularité
    - Différents niveaux d'automatisation

Tests automatisés

Niveaux de tests informatisés

- **Tests unitaires**: vérifient chaque classe ou module individuellement
- **Tests d'intégration**: vérifient les interactions entre plusieurs classes ou modules
  - Souvent utilisés lors de l'ajout d'un nouveau module à un groupe de modules (testés) existants
- **Tests de régression**: vérifient qu'un changement (ex: correction) n'a pas introduit de nouvelles erreurs
  - En théorie, tous les tests de régression devraient être exécutés après chaque changement
  - En pratique, exécuter tous les tests est trop coûteux
  - Certains outils permettent de choisir un sous-ensemble de tests à exécuter

Niveaux de tests informatisés

- **Tests de système**: vérifient que le système en entier est conforme aux spécifications
- **Tests de validation**: s'assurent que le système réponde aux besoins du client
- **Tests de marges** (stress tests): contrôlent la performance et la fiabilité d'un système dans des conditions similaires aux conditions normales d'opération
  - Ex: Augmenter graduellement le nombre de transactions d'un système distribué

Tests fonctionnels

- Aussi appelés tests « boîte noire » (black box tests)
- Identifient les fautes en se basant uniquement sur les entrées et les sorties
- Ne tiennent pas compte du comportement interne du programme
  - Ex: chemin exécuté pour le calcul, cas spéciaux
- La spécification et/ou les connaissances du domaine sont utilisées pour déterminer les cas à tester

Tests fonctionnels

- Avantages
  - Supportent plusieurs types de tests
  - Requièrent moins de ressources
- Inconvénients
  - Ne permettent pas d'identifier les cas où plusieurs erreurs s'annulent pour produire le bon résultat
  - Ne permettent pas d'évaluer la couverture du code
  - Ne permettent pas d'évaluer la qualité du code

Tests structurels

- Aussi appelés tests « boîte blanche » (white box tests) ou « boîte de verre » (glass box tests)
- Exploitent la connaissance de la structure du code
- Requièrent parfois des analyses pour s'assurer que le code a été couvert adéquatement
  - Par ligne de code
  - Par instruction
  - Par branche
  - …

Tests structurels

- Avantages
  - Permettent de vérifier l'implémentation d'un algorithme de façon précise
  - Permettent d'évaluer la quantité de code non-testé
  - Permettent d'évaluer la qualité du code et son niveau d'adhérence aux standards
- Inconvénient
  - Utilisent plus de ressources

Tests exhaustifs

- Est-il possible de tester un programme complèment?
  - c'est-à-dire, trouver toutes les erreurs présentes
- Tests fonctionnels :
  - Pour tester le programme complètement, il faudrait tester toutes les entrées possibles
  - Une infinité de cas en général

Tests exhaustifs

- Tests structurels
  - Programme de ~ 50-100 lignes
  - 1 boucle (1-20 itérations), 4 conditions imbriquées

```
do
   if (…) then
       if (…) then
          if (…) then … else …
       else …
          if (…) then … else …
   else …
while (i < 20)
```

  - ~ 10^{14} chemins d'exécution possibles ($\sum_{i=1}^{20} 5^i$)
  - 3174 ans pour tester tous les chemins à raison d'un chemin par milliseconde!

Tests exhaustifs

- En général, tester un programme de façon exhaustive est impossible
- Il faut choisir un sous-ensemble des tests qui maximise la probabilité de détecter les erreurs
- Il est possible d'utiliser des tests aléatoires, mais leur efficacité est faible pour tester le comportement attendu
  - « Fuzz testing » est une technique répandue qui consiste à tester les cas inattendus ou anormaux à l'aide de valeurs d'entrée générées aléatoirement, et permet d'augmenter la robustesse d'un programme
- Une meilleure approche : déterminer un ensemble de tests fonctionnels qui seront complémentés de tests structurels

Tester en présence de dépendances

- La plupart des modules requièrent d'autres modules pour leur bon fonctionnement
- Idéalement, chaque module devrait être testé en isolation
  - Débogage plus facile
  - Tests plus robustes en présence de changements dans le système
  - Permet une meilleure organisation des tests
- Comment isoler le comportement d'un module particulier?
  - Par la création d'un « faux module » synthétique qui joue le rôle de la dépendance pour les tests effectués
  - Par exemple, le faux module peut retourner des valeurs prédéterminées lorsqu'invoqué
  - (plus de détails à venir dans la prochaine section)

JUnit

Tests unitaires

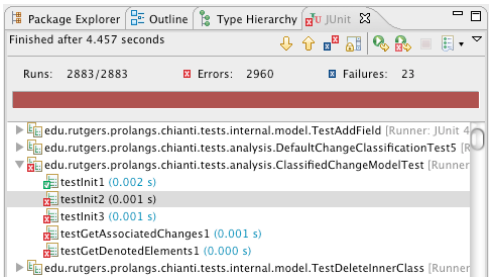
- Un test unitaire vise à tester une unité individuelle d'un programme
  - En général, une fonction, une méthode, une classe ou un module
- But : comparer le résultat d'une opération à sa spécification
  - Il faut tenter de démontrer que l'unité contredit sa spécification
- Motivation :
  - Permet d'identifier les erreurs plus facilement lorsqu'une partie restreinte du code est testée
    - La source de l'erreur doit se trouver dans l'unité testée
  - Permet de tester plusieurs unités en parallèle
  - Permet de tester une unité lorsque le système est encore incomplète

Écrire des tests unitaires

- Les tests unitaires sont souvent structurels
  - Comme les tests unitaires sont exécutés sur une portion très restreinte du code, les tests structurels sont beaucoup moins coûteux à effectuer qu'aux autres niveaux de tests
  - Les tests d'ordre supérieurs tentent souvent d'identifier d'autres types d'erreurs que les erreurs de programmation et de logique détectées par les tests structurels
- Les techniques de génération de tests peuvent être utilisées pour déterminer un ensemble de tests unitaires
- En TDD, les tests fonctionnels sont initialement générés avant de commencer l'implémentation
  - Les tests structurels sont ajoutés au fur et à mesure que l'implémentation progresse

Outils

- Les tests unitaires sont souvent automatisés
  - Un outil se charge d'exécuter tous les tests en séquence, et présente les résultats de façon visuelle



JUnit

- Outil automatisé permettant de tester des programmes Java
  - Permet d'exécuter un ou plusieurs cas de test (test cases) automatiquement
  - Le jugement d'un utilisateur n'est pas requis
- Intégré à la plupart des IDEs
- Porté à la grande majorité des langages les plus populaires

Concepts

- Cas de test (*test case*):
  - Les tests sont en général indépendants
  - L'état doit être recréé pour chaque test
- Fixture: une collection de cas de test qui testent une seule classe du système
  - Une fixture permet de définir des objets, mais les objets sont recréés pour chaque test
- Suite de tests (*test suite*): une collection de fixtures qui teste toutes les classes du système

JUnit – Syntaxe


```
public class TestSuite {
    @Test
    public void testSomething() {
        // test
    }
}
```

Pour exécuter les tests :

```
org.junit.runner.JUnitCore.runClasses(TestSuite.class)
```

(ou utiliser l'intégration avec Eclipse, par exemple)

Assertions

- Méthodes statiques déclarées dans la classe `org.junit.Assert`
  - `assertEquals(message, expected, actual)`
  - `assertTrue(message, condition)`
  - `assertFalse(message, condition)`
  - `assertNull(message, object)`
  - `assertNotNull(message, object)`
  - `assertSame(message, expected, actual)`
  - `assertNotSame(message, expected, actual)`

JUnit – Tester une valeur de retour


```
public class IntTests {
    @Test
    public void testParse() {
        int v = Integer.parseInt("-FF", 16);
        Assert.assertEquals(-255, v);
    }
}
```

Assertions - Égalité vs comparaison

- Préférer `assertEquals` à `assertTrue`
  - Avec assertEquals :

```
org.junit.ComparisonFailure: fib(3) expected: <3> but was: <2>
```

  - Avec assertTrue :

```
java.lang.AssertionError: fib(3)
```


Assertions - Nombres à virgule

- Pour comparer des nombres à virgule, il est préférable de spécifier une tolérance pour éviter les problèmes d'instabilité des calculs :

```
assertEquals("Velocity", 1.23, velocity);
assertEquals("Velocity", 1.23, velocity,
  0.001 // tolerance
);
```


JUnit – Exemple


```
public class CollectionTests {
    @Test public void emptyCollection() {
        Collection collection = new ArrayList();
        assertEquals(0, collection.size());
        assertTrue(collection.isEmpty());
    }

    @Test public void addOneItem() {
        Collection collection = new ArrayList();
        collection.add("itemA");
        assertEquals(1, collection.size());
        assertTrue(collection.contains("itemA"));
    }
```


Ignorer un test temporairement


```
public class CollectionTests {
    @Ignore @Test public void emptyCollection() {
        Collection collection = new ArrayList();
        assertEquals(0, collection.size());
        assertTrue(collection.isEmpty());
    }

    @Test public void addOneItem() {
        Collection collection = new ArrayList();
        collection.add("itemA");
        assertEquals(1, collection.size());
        assertTrue(collection.contains("itemA"));
    }
```

JUnit – Tester une exception


```
public class IntTests {
    @Test(expected=NumberFormatException.class)
    public void testParseException() {
        int v = Integer.parseInt("abc");
    }

    @Test
    public void testParseExceptionAlternate() {
        try {
            int v = Integer.parseInt("abc");
            Assert.fail("No exception thrown");
        } catch (NumberFormatException e) {
            // pass
        }
    }
}
```


Autres méthodes supportées

- `@Before public void setUp()`
  - `setUp` est appelée avant l'exécution de chaque test
- `@After public void tearDown()`
  - `tearDown` est appelée après l'exécution de chaque test
- `@BeforeClass public static void oneTimeSetUp()`
  - `oneTimeSetUp` appelée avant de débuter l'exécution du premier test
- `@AfterClass public static void oneTimeTearDown()`
  - `oneTimeTearDown` est appelée après que tous les tests aient été effectués

Exemple


```
public class SimpleTest {
    private Collection<Object> collection;

    @Before public void setUp() {
        collection = new ArrayList<Object>();
    }

    @Test public void testEmptyCollection() {
        assertTrue(collection.isEmpty());
    }

    @Test public void testOneItemCollection() {
        collection.add("itemA");
        assertEquals(1, collection.size());
    }
}
```


JUnit – Limites de temps


```
public class IntTests {
    @Test(timeout=100) // délai permis de 100 ms
    public void testParseException() {
        int result = fib(30);
        Assert.assertEquals("fib(30)",
            1346269,
            result
        );
    }
}
```

JUnit - Suites de tests


```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    LexerTests.class,     // Les classes définissant les tests à
    ParserTests.class,    // regrouper
    ASTTests.class,       //
})
public class AllTests {
   // class vide, tout le travail est effectué par les annotations
}
```


Execution - Ligne de commande


```
java org.junit.runner.JUnitCore <suiteOuFixture>
```

Par exemple :

```
java org.junit.runner.JUnitCore compiler.LexerTests
java org.junit.runner.JUnitCore FrontEndSuite
```


Execution - Eclipse

Sélectionner une fixture ou une suite, puis :

`Run As… > JUnit Test Case`

Organisation

- Garder les classes de tests dans le même projet que le code
  - Les test sont compilés avec le reste du code
  - Aide à actualiser les tests
- Grouper les tests dans le même paquetage, mais un dossier différent des fichiers source
  - ex: `src/` et `tests/`
  - Permet aux tests d'accéder aux entités visibles seulement dans leur paquetage
- Utiliser une nomenclature descriptive :
  - ex: `ParserTest` teste la classe `Parser`

Quoi tester?

- Valeurs “normales”
  - ex: valeurs aléatoires raisonnables
- Les cas limites
  - ex: 0, Integer.MAX_VALUE, tableau vide
- Valeurs inattendues
  - ex: null, caractères invalides dans une chaîne
- Différentes catégories d’entrées
  - ex: entier positif, négatif, et zéro
- Différents comportements possibles
  - ex: chaque message d’erreur, toutes les options d’un menu

FAQ

- Comment tester les méthode privées?
  - En général, elles ne devraient pas être testées directement, mais c’est tout de même possible de les tester à l’aide de mécanismes de réflection.
- Doit-on tester tous les “getters” et “setters”?
  - En général, c’est inutile, il suffit donc de tester seulement les getters et setters qui sont susceptibles d’être défectueux.
- Comment tester des classes abstraites?
  - À l’aide de classes abstraites contenant des tests qui sont hérités par les fixtures concrètes.

Démonstration