IFT3912 - Développement et maintenance de logiciels

Tests automatisés

Bruno Dufour
dufour@iro.umontreal.ca

Terminologie

- Système sous test, ou SST (*System Under Test - SUT*)
  - Un système ou une partie du système qui est en train d'être testé pour s'assurer de son bon fonctionnement
- Tests automatisés
  - Tests qui exécutent le SST pour observer son comportement et déterminer sa validité

Qualités de tests

- Les tests automatisés devraient être
  - faciles à exécuter
    - vérifiés sans intervention manuelle (*self-checking*)
    - reproductibles
  - Simples à écrire
    - Ne testent pas trop de fonctionnalité à la fois
    - Utilisent des abstractions appropriées
  - Simples à maintenir
    - Affectés seulement par des changements reliés à la fonctionnalité directement testée

Philosophie

Tester avant ou après ?

- Écrire les tests après le code à tester
  - Difficile, peut nécessiter des modifications à du code “déjà complété”
- Écrire les tests avant le code à tester (*TDD*)
  - permet de réduire l’effort de débogage
  - force le code à être écrit de façon à pouvoir être testé facilement
  - minimise la quantité de code développé
  - rend les tests plus robustes puisqu’ils ne dépendent habituellement que des méthodes strictement nécessaires
  - fournit des exemples d’utilisation (cible concrète)

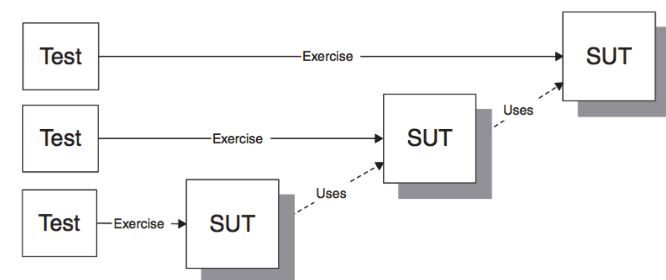
Un ou plusieurs tests à la fois ?

- Écrire un test à la fois
  - Simplifie ou évite souvent le débogage
  - Permet de se concentrer sur un changement à la fois
- Écrire plusieurs tests à la fois avant d’écrire le code à tester
  - Permet de réfléchir en “mode client”
  - Retarde le passage en mode solution
  - Peut être utilisé pour produire un plan initial des tests à écrire

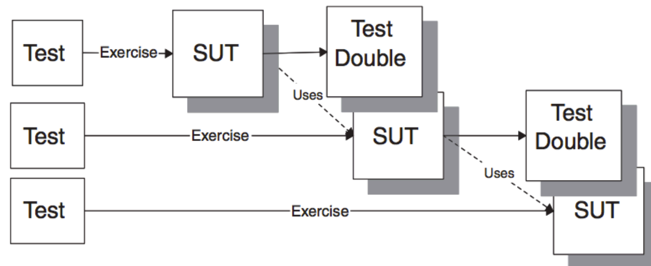
Débuter par l’extérieur ou l’intérieur ?

- De l’extérieur vers l’intérieur
  - Débute par les modules de haut niveau
  - Permet d’adopter le point de vue du client avant celui du programmeur
  - Oblige à gérer les dépendances manquantes
- De l’intérieur vers l’extérieur
  - Débute par les modules de bas niveau
  - Évite les problèmes de dépendances
  - Ne permet pas de tester les modules de haut-niveau indépendamment du reste du système

Débuter par l’intérieur



Débuter par l'extérieur



Principes

Concevoir pour faciliter les tests

- Du code conçu sans se soucier des tests peut
  - rendre difficile l'exercice d'une partie spécifique du code
  - compliquer l'accès à l'état du SST
- Facilité par l'écriture de tests avant le code à tester

Préférer la porte principale

- Préférer l'utilisation de l'interface publique d'une classe pour les tests
  - Permet d'obtenir des tests plus robustes
  - Moins sujet au changement
- La surutilisation de vérification du comportement peut mener au même problème

Ne pas modifier le SST

- Remplacer du code du SST par une version synthétique est une pratique commune
  - remplacer des dépendances non-satisfaites
  - éviter les effets de bords inacceptables
  - prendre contrôle des entrées et sorties indirectes
- Il ne faut jamais tester ces remplacements!
  - Le SST doit toujours exclure le code synthétique

Garder les tests indépendants

- Si deux tests sont interdépendants, il est possible qu'un test échoue seulement parce que l'autre test a aussi échoué
  - Beaucoup de projets contiennent des centaines de tests
  - Résultats impossibles à interpréter si des dépendances existent

Isoler le SST

- Lorsque le SST dépend d'autres composants, des changements ceux-cis peuvent faire échouer les tests
  - fragilité des tests
- Les dépendances devraient pouvoir être remplacées par des doublures synthétiques lors des tests
  - test plus robustes
  - facilité par l'injection de dépendances

Minimiser la redondance

- Plusieurs tests redondants
  - n'améliorent pas la qualité de l'ensemble des tests
  - requièrent plus d'effort à écrire et à maintenir
- Chaque test doit offrir une contribution qui lui est propre
  - Chaque condition à tester devrait être couverte par précisément 1 test

Minimiser le code non-testé

- Durant le développement, certaines parties du code se retrouvent dans des contextes qui les rendent difficiles à tester
  - ex: GUI, code concurrent, etc.
- La logique du code peut être déplacée vers un autre environnement plus facile à tester
  - ex: découpler la logique d'un composant graphique de sa représentation visuelle

Séparer la logique de test

- Les développeurs sont souvent tentés par la possibilité d'ajouter des "hooks" au code de production pour le rendre plus facile à tester
  - Souvent, `if (testing) then ...`
    - Cause un comportement différent lors d'une exécution de test comparativement à une exécution normale
    - Rend très difficile l'observation du comportement
    - Peut même causer des défaillances dans le code de production
  - Le code de production ne devrait jamais contenir de logique de test

Ne vérifier qu'une condition par test

- Les tests automatisés peuvent être exécutés en groupe et rapidement
  - Contrairement aux tests manuels, les test automatisés peuvent être très ciblés
- Un test ciblé permet d'identifier la cause du problème détecté plus efficacement
  - Moins d'état à inspecter
  - Aucun flot de contrôle conditionnel

Exemple


```
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
    LineItem expected = new LineItem(invoice,
                            product,
                            5,
                            new BigDecimal("30"),
                            new BigDecimal("69.96"));
    LineItem actItem = (LineItem) lineItems.get(0);
    assertEquals("invoice", expected, actItem);
} else {
    fail("Invoice should have exactly one line item");
}
```

xUnit

xUnit

- xUnit désigne une famille d'outils qui visent à
  - permettre aux programmeurs d'écrire des tests automatisés dans le langage de programmation de leur choix
  - permettre de tester une classe ou méthode sans que le reste d'un système soit disponible
  - permettre l'exécution d'un test ou plusieurs tests à l'aide d'une seule action
  - minimiser le coût associé à l'écrire de tests de façon à répandre cette pratique

xUnit - concepts communs

- Chaque test est décrit par une **méthode de test**
- Les résultats attendus des tests sont spécifiés à l'aide d'appels à des **méthodes d'assertion**
- Les tests peuvent être groupés dans des **suites de tests** pour faciliter leur exécution
- Un **lanceur de tests** (*test runner*) permet d'exécuter une suite de tests automatiquement et d'afficher les résultats de l'exécution

Exemple - jUnit

Test suite

Test case

```
public class IntTests {
    @Test
    public void testParseDec() {
        int v = Integer.parseInt("-255", 10);
        Assert.assertEquals(-255, v);
    }

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

Assertion (résultat attendu)

Exemple - Mocha (JavaScript)

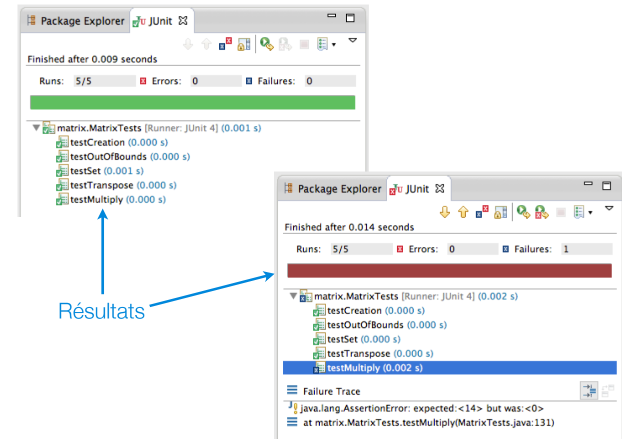
Test suite

```
describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present',
    function() {
      [1,2,3].indexOf(5).should.equal(-1);
      [1,2,3].indexOf(0).should.equal(-1);
    })
  })
})
```

Assertion

Test case

Exemple - jUnit - Test runner



Fixture

- Fixture : ce qui est nécessaire pour exécuter un test
  - Une partie du SST initialisée convenablement
  - Habituellement, comprend au moins une instance de la classe en train d'être testée
  - Peut comprendre d'autres objets selon la fonctionnalité exercée par le test
- Chaque test devrait utiliser une nouvelle fixture de façon à préserver l'indépendance des tests entre eux
  - Chaque fixture ainsi créée n'est utilisée qu'une seule fois puis détruite
  - Les tests devraient pouvoir être exécutés dans n'importe quel ordre

Exemple - Fixture (*inline*)

Fixture

```
@Test
public void testStatus initial() {
    Airport departureAirport = new Airport("YUL");
    Airport destinationAirport = new Airport("YYZ");
    String flightNumber = "AB123";
    Flight flight = new Flight(flightNumber,
                               departureAirport,
                               destinationAirport);

    assertEquals(FlightState.PROPOSED, flight.getStatus());
}
```

Exemple - Fixture (déléguée)


```
public Flight createFlight() {
    Airport departureAirport = new Airport("YUL");
    Airport destinationAirport = new Airport("YYZ");
    String flightNumber = "AB123";
    return new Flight(flightNumber,
                            departureAirport,
                            destinationAirport);
}

@Test
public void testStatus_initial() {
    Flight flight = createFlight();


    assertEquals(FlightState.PROPOSED, flight.getStatus());
}
```


Exemple - Fixture (implicite)


```
Flight flight;


@Before
public void setUp() {
    Airport departureAirport = new Airport("YUL");
    Airport destinationAirport = new Airport("YYZ");
    String flightNumber = "AB123";
    flight = Flight(flightNumber,
                            departureAirport,
                            destinationAirport);
}

@Test
public void testStatus_initial() {
    assertEquals(FlightState.PROPOSED, flight.getStatus());
}
```


Fixture éphémère (*transient*)

- Une fixture est **éphémère**
  - Utilisée une seule fois
  - Est réclamée automatiquement à la fin du test
- Une fixture peut devenir **permanente** si
  - le code de test garde une référence à la fixture
  - l'état du SST est affecté et cette modification persiste entre les tests
    - ex: modification d'une base de données, attributs de classes
- Une fixture permanente peut entraîner des tests non-reproductibles

Exemple - Fixture éphémère ?


```
Flight flight;


@Before
public void setUp() {
    Airport departureAirport = new Airport("YUL");
    Airport destinationAirport = new Airport("YYZ");
    String flightNumber = "AB123";
    flight = Flight(flightNumber,
                            departureAirport,
                            destinationAirport);
}

@Test
public void testStatus_initial() {
    assertEquals(FlightState.PROPOSED, flight.getStatus());
}
```

Exemple - Fixture éphémère ?


```
@Before
public void setUp() {
    Airport departureAirport = new Airport("YUL");
    Airport destinationAirport = new Airport("YYZ");
    String flightNumber = “AB123”;
    flight = Flight(flightNumber,
                               departureAirport,
                               destinationAirport);
}

@Test
public void testStatus_initial() {
    assertEquals(FlightState.PROPOSED, flight.getStatus());
}

@After
public void tearDown() {
    flight = null;
}
```


Exemple - Fixture permanente


```
@Test
public void testGetFlightsByOriginAirport_NoFlights() {
    // Fixture Setup
    Airport outboundAirport = facade.createTestAirport("YUL");
    List<?> flights = facade.getFlightsByOriginAirport(
        outboundAirport);


    assertEquals(0, flights.size());
}
```


Exemple - Fixture permanente


```
@Test
public void testGetFlightsByOriginAirport_NoFlights() {
    try {
        // Fixture Setup
        Airport outboundAirport =
            facade.createTestAirport("YUL");
        List<?> flights = facade.getFlightsByOriginAirport(
            outboundAirport);


        assertEquals(0, flights.size());
    } finally {
        facade.removeAirport(outboundAirport);
    }
}
```


Exemple - Fixture permanente


```
@Test
public void testGetFlightsByOriginAirport_NoFlights() {
    // Fixture Setup
    Airport outboundAirport = facade.createTestAirport("YUL");
    List<?> flights = facade.getFlightsByOriginAirport(
        outboundAirport);


    assertEquals(0, flights.size());
}

@After
public void tearDown() {
    facade.resetAirports();
}
```

jUnit - Rule

Ce dossier sera effacé automatiquement après l'exécution de chaque test

```
@Rule
public TemporaryFolder folder = new TemporaryFolder();

@Test
public void testUsingTempFolder() throws IOException {
    File createdFolder = folder.newFolder("newfolder");
    File createdFile = folder.newFile("myfilefile.txt");
    assertTrue(createdFile.exists());
}
```


Éviter la permanence d'une fixture

- Souvent, il est préférable de modifier le code des tests pour éviter la permanence
  - ex: utiliser une base de données en mémoire plutôt qu'une base de données sur le disque, et réinitialiser pour chaque test
  - les tests sont souvent plus performants en évitant d'accéder au disque de façon répétée
  - approche générale : remplacer une partie du SST par une doublure qui évite la permanence
- On peut aussi s'assurer que les changements d'états visibles ne causent pas de collision entre les tests

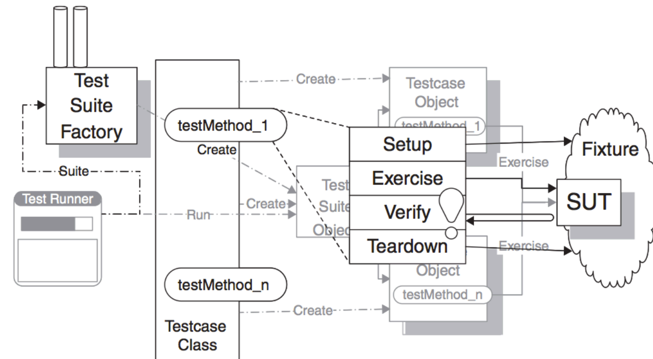
Fixture partagée (*shared*)

- Dans certains cas, il peut être préférable de ne pas recréer une fixture pour chaque test, par exemple :
  - lorsque la fixture coûtent trop cher à recréer
  - lorsque la fixture est immuable
  - lorsqu'on cherche à tester une longue séquence d'opérations interdépendantes (le plus souvent dans des tests de système)
  - lorsqu'une fixture par exécution (*per-run*) suffit

Phases d'exécution

- L'exécution de chaque test procède en 4 étapes
  - *setup* : crée d'une fixture
  - *exercise* : interagit avec le SST
  - *verify* : détermine si le résultat attendu a été obtenu
  - *teardown* : annule les changements d'état (retourne à l'état initial)

Structure des tests



Stratégie

Granularité

- **Unité**
  - ex: classe, méthode
  - les tests unitaires permettent de documenter le comportement attendu d'une unité
- **Composant**
  - un groupe de classes qui fournissent un service collectivement
- **Système**
  - Par les développeurs : tests de système
  - Par le clients : tests d'acceptation

Autres types de tests

- **Tests d'intégration** : vérifie les interactions entre plusieurs classes lors de l'ajout d'un nouveau classe à un groupe de classes (testées) existantes
- **Tests de régression**: vérifient qu'un changement préserve le comportement existant
- **Tests de performance** : contrôlent la performance et la fiabilité d'un système dans diverses conditions, ex:
  - **load tests** : conditions similaires aux conditions normales d'opération
  - **stress tests** : conditions qui excèdent les conditions normales
  - **capacity tests** : visent à déterminer la charge supportée par un système

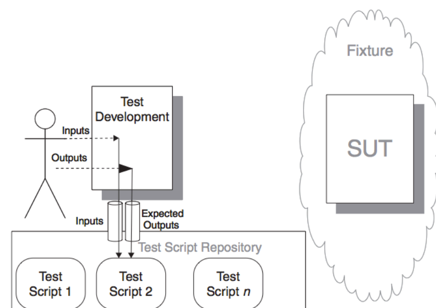
Autres types de tests

- **Tests d'injection de fautes** : permettent le comportement d'un système en cas d'erreur
  - ex: défaillance de système de fichier ou d'un périphérique
- **Tests d'utilisation** : permettent de vérifier qu'un système remplit bien ses objectifs
  - difficiles à automatiser, implique souvent des utilisateurs

Stratégie

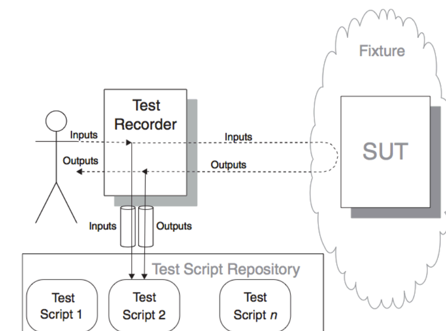
- Tests scriptés
- Tests préenregistrés (*record-replay*)
- Tests pilotés par les données (*data-driven*)

Tests scriptés



- Utilisés pour spécifier le comportement attendu du SST avant que le code ne soit disponible

Tests enregistrés



- Utilisés pour générer rapidement des tests de régression à partir d'un SST fonctionnel

Tests pilotés par les données

- Utilisés pour simplifier la tâche d'écriture des tests
  - Automatisent une tâche répétitive
  - Peut faciliter la maintenance tout en offrant une excellente couverture du code
- Cette stratégie peut servir à rejouer des tests préalablement enregistrés
- Support natif pour la paramétrisation des tests dans jUnit

jUnit - Tests paramétrés


```
@RunWith(Parameterized.class)
public class PrimeNumberCheckerTest {
   ...

   @Parameterized.Parameters
   public static Collection primeNumbers() {
      return Arrays.asList(new Object[][] {
         { 2, true }, { 6, false }, { 19, true }, { 22, false }, { 23, true }
      });
   }

   public PrimeNumberCheckerTest(Integer n, Boolean expectedResult) {
      this.inputNumber = n;
      this.expectedResult = expectedResult;
   }

   // Exécuté 5 fois
   @Test
   public void testPrimeNumberChecker() {
      assertEquals(expectedResult, primeNumberChecker.validate(inputNumber));
   }
}
```


Vérification

Vérification des résultats

- 2 stratégies
  - Vérification de l'état (TDD)
    - Extraction de l'état du SST à partir de points d'observation
    - Comparaison avec l'état attendu
  - Vérification du comportement (BDD)
    - Placement de points d'observation entre le SST et les composants dont il dépend
    - Comparaison de la séquence d'appels avec le comportement attendu

Assertions

- Assertions de résultat, assertTrue
- Assertions d'égalité simple : assertEquals(expected, actual)
- Assertions approximatives : assertEquals(expected, actual, tolerance)
- Assertions personnalisées
- Méthodes de vérification
  - Font appel à une ou plusieurs autres assertions
  - Manipule le SST

Objets attendus

- Un test doit souvent comparer plusieurs attributs du même objet
  - On peut utiliser l'égalité entre objets pour faire abstraction de cette vérification

Exemple - Objet attendu


```
public void testInvoice_addLineItem7() {
      LineItem expItem = new LineItem(inv, product, QUANTITY);

      inv.addItemQuantity(product, QUANTITY);

      List lineItems = inv.getLineItems();
      LineItem actual = (LineItem) lineItems.get(0);

      assertEquals(expItem.getInv(), actual.getInv());
      assertEquals(expItem.getProd(), actual.getProd());
      assertEquals(expItem.getQuantity(),
          actual.getQuantity());
}
```


Exemple - Objet attendu


```
public void testInvoice_addLineItem7() {
      LineItem expItem = new LineItem(inv, product, QUANTITY);

      inv.addItemQuantity(product, QUANTITY);

      List lineItems = inv.getLineItems();
      LineItem actual = (LineItem) lineItems.get(0);

      assertEquals("Item", expItem, actual);
}
```

Objets attendus

- En pratique, le constructeur de l'objet peut ne pas être accessible
  - Utiliser une assertion personnalisée à la place d'utiliser equals(Object)
  - Créer une classe spécifique pour effectuer la comparaison
    - Sa méthode equals traitera le type de l'objet comparé

Exemple - Assertion personnalisée


```
static void assertLineItemsEqual(String  msg, LineItem exp,
        LineItem act) {
    assertEquals(msg+" Inv",  exp.getInv(), act.getInv());
    assertEquals(msg+" Prod", exp.getProd(), act.getProd());
    assertEquals(msg+" Quan", exp.getQuantity(),
        act.getQuantity());
}
```


Exemple - Méthode de vérif.


```
void assertInvoiceContainsOnlyThisLineItem(
                                     Invoice inv,
                                     LineItem expItem) {
      List lineItems = inv.getLineItems();
      assertEquals("number of items", lineItems.size(), 1);
      LineItem actual = (LineItem)lineItems.get(0);
      assertLineItemsEqual("",expItem, actual);
}
```