ENSEIRB
Année 2013-2014

3ème année
IT306

# TD n°2 - Techniques de test et programmation objet

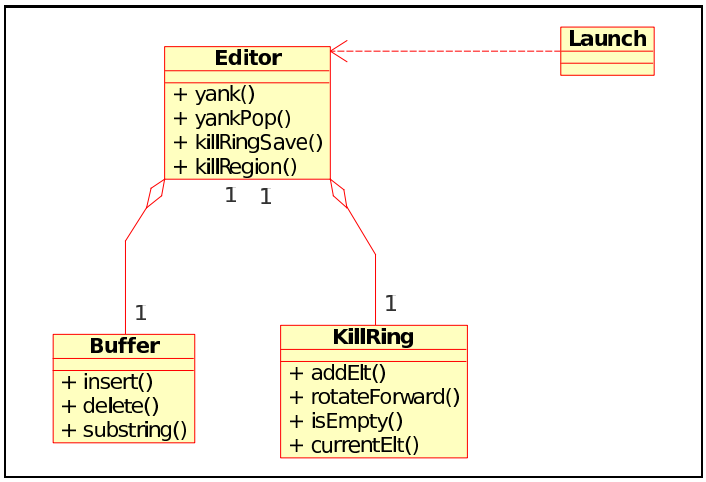
**Résumé** : Cette feuille s'intéresse au test de composants dans un cadre modulaire (ici appliqué à la programmation objet). Les tests sont décomposés en deux familles : les tests unitaires d'une part, et les tests d'intégration d'autre part.

⚠ Pour pouvoir utiliser les archives Java du TD, il est nécessaire d'utiliser une version spécifique du `JDK` supérieure ou égale à la version 1.6. Pour cela, il suffit de vérifier alors que la commande `java -version` renvoie un nombre commençant au moins par `"1.6"`.

*Remarque* : dans un premier temps, il n'est pas demandé de réaliser toutes les classes de test, mais plutôt de privilégier d'arriver à la fin de la feuille pour terminer les tests ensuite.

**Exercice 1: Prise en main de l'architecture Editor**

Considérons le diagramme de classes suivant, représentant l'architecture d'un éditeur de texte permettant d'effectuer des opérations de style copier/couper/coller sur les tampons (*buffers*) qu'il manipule. Les chaînes de caractères ainsi manipulées sont stockées dans une liste circulaire appelée *killring*[1].




Dans notre implémentation :

- les buffers sont stockés dans une classe `Buffer` sous forme d'une chaîne de caractères ;
- le killring correspond à la classe `KillRing` et est réalisé sous la forme d'une `LinkedList` ;
- le killring ne peut contenir plus de `KillRing.TAILLE_MAX` éléments, sous peine de lancer une exception `BufferOverflowException`.

1. Toute ressemblance dans cet exercice avec un éditeur de fichiers existant ou ayant jamais existé ne serait qu'une vulgaire coïncidence. Néanmoins, rappellons que sous Emacs, il est possible d'obtenir de l'aide sur n'importe quelle fonction en tapant `[C-h] [f]`

0. Exécuter la classe Launch et vérifier qu'elle effectue une utilisation nominale de l'éditeur : instanciation, copie d'une zone de texte, découpe d'une zone, et deux insertions successives au même endroit.

**Exercice 2: Test unitaire avec JUnit**

Pour écrire des tests en Java, nous allons utiliser la bibliothèque JUnit4 (`http://junit.sourceforge.net/`). Cette bibliothèque[2] promeut le développement du code en même temps que des tests de ce code. Elle a été déclinée dans un nombre important de langages : CUnit, CPPUnit, AUnit (Ada), HUnit (Haskell), XMLUnit et d'autres encore.

▷ Afin de tester une classe avec JUnit, il faut commencer par écrire une classe de test. JUnit4 utilise les annotations Java : il suffit de rajouter le mot clé @Test devant une méthode **public void** pour que le *runner*[3] de JUnit la considère comme un test à réaliser :

```
1  import org.junit.Test ;
2  import static org.junit.Assert.* ;
3
4  public class LocomotiveTest {
5
6      @Test public void testChimney() {
7          Locomotive loco = new Locomotive();
8          assertTrue(loco.chimney.smokes());
9          assertTrue(loco.getName().equals("Salamanca"));
10     }
11 }
```

La méthode assertTrue sert à JUnit pour déterminer si le test a réussi ou échoué. Plusieurs assertions peuvent être vérifiées dans un même test. La documentation de JUnit est disponible à l'adresse `http://junit.sourceforge.net/javadoc_40/index.html`.

Remarquons tout d'abord que dans notre architecture, il y a trois classes à tester, et qu'il faut les tester indépendamment. Pour l'instant, on ne s'intéresse qu'au test des classes Buffer et KillRing qui sont les feuilles du diagramme de classe.

1. Écrire une classe de test BufferTest vide pour la classe Buffer, puis implémenter un test simple de la méthode toString() à l'intérieur.

   Utiliser la règle test du Makefile pour vérifier qu'il passe correctement.

2. Comment donner un nom à une méthode de test en général ? Plus généralement, quelles sont les qualités que l'on peut attendre de code contenu dans les méthodes de test ?

2. Développée par entre autres E. Gamma (Design Patterns) et K. Beck (Test Driven Develoment)
3. Le *runner* est le nom de la classe de JUnit dont le rôle consiste à exécuter les tests.

**Exercice 3: Visualisation de la couverture**

JUnit n'est pas un outil de mesure de couverture sur le code. Pour vérifier que les tests codés sont suffisamment exhaustifs, il est nécessaire d'utiliser un outil annexe. Dans cet exercice, on utilise Cobertura (`http://cobertura.sourceforge.net/`), un outil qui instrument le code Java et génère des rapports HTML.

1. Calculer avec Cobertura le taux de couverture assuré par vos classes de test. Pour cela, utiliser la règle report du fichier Makefile, et jeter un coup d'oeil aux fichiers produits dans le sous-répertoire `report/`.
2. Quels sont les types de tests structurels pris en charge par Cobertura ?

Dans la suite de cette feuille, on vérifiera la couverture des tests réalisés en utilisant ce logiciel.

**Exercice 4: De l'écriture des tests**

Écrire des programmes de tests n'est pas une activité naïve : il faut veiller à ce que l'on n'ait pas à tester aussi les tests ! À cet effet, il est nécessaire d'écrire les méthodes de test de manière à ce que leur compréhension soit la plus directe possible.

▷ JUnit embarque une bibliothèque nommée Hamcrest (cf. `http://code.google.com/p/hamcrest/`), qui contient un nombre important de tests d'assertion. La méthode d'assertion générique se nomme assertThat :

```
1 import static org.hamcrest.MatcherAssert.assertThat;
2 import static org.hamcrest.Matchers.*;
3
4 assertThat(buff.substring(11,14), is(equalTo("est")));
```

L'intérêt de Hamcrest provient principalement de la possibilité d'utiliser de nombreux tests d'assertion (plus riches que JUnit seul), et d'augmenter la lisibilité des tests (dans le fichier de test et dans les messages d'erreur). La documentation est disponible à l'adresse `http://junit.sourceforge.net/javadoc/org/hamcrest/core/package-summary.html`.

1. Quel est le comportement et l'intérêt d'un test d'assertion comme Is<T> ?

Imaginons que notre éditeur de fichiers puisse manipuler plusieurs types de Buffer différents. Par exemple, un ReadOnlyBuffer renvoie une exception lorsque l'on demande de supprimer ou d'insérer du texte à l'intérieur.

2. Comment modifier le diagramme de classes pour pouvoir prendre en compte une telle organisation des Buffer sans modifier la classe Editor ?
3. Les deux classes Buffer et ReadOnlyBuffer possèdent une partie de leur code en commun. Comment peut-on faire pour factoriser les tests qui leur correspondent ? Quels problèmes cela peut-il engendrer ?
4. Terminer de tester la classe BufferTest pour obtenir une couverture $\geq 80\%$.

**Exercice 5: Gestion des exceptions**

La classe KillRing a un comportement légèrement plus complexe que la classe Buffer : la méthode addElt peut renvoyer une exception. Néanmoins, en Java, même si les exceptions ont tendance à désorganiser le graphe de flot de contrôle, elles ne posent pas de problèmes pour être testées.

1. Écrire une classe de test KillRingTest contenant un test permettant de gérer l'exception BufferOverflowException lancée lorsque le KillRing est plein. Pour cela, on utilisera le mécanisme de JUnit4 permettant de spécifier qu'un test doit renvoyer une exception :

```
1 @Test(expected= BufferOverflowException.class)
```

⚠ Penser à importer dans vos tests l'exception en question en incluant la ligne "**import** java.nio.BufferOverflowException" dans vos en-têtes.

Pour pouvoir lever cette exception, il est nécessaire de remplir suffisamment la liste elements de l'objet KillRing. Si l'on utilise une boucle pour cela, il reste toujours la possibilité de se tromper en écrivant la boucle.

2. Comment écrire un test clair permettant de s'assurer que le nombre d'ajouts nécessaires à lever cette exception soit bien KillRing.TAILLE_MAX ?
3. Est-il nécessaire de tester les méthodes privées ? Si oui, comment le fait-on ?
4. Terminer de tester la class KillRingTest pour obtenir une couverture $\geq 80\%$.

**Exercice 6: Test de spécification avec QuickCheck**

Dans la plupart des tests écrits ci-dessus, les données de test sont spécifiées de manière statique. Une méthode de test va mettre en place un seul et unique jeu de test, qu'elle va exécuter. Même si les méthodes de test structurel valident ce type d'approche, il est possible d'adopter une méthode plus générique.

▷ La bibliothèque QuickCheck (disponible à l'adresse `http://java.net/projects/quickcheck/pages/Home`), basée sur une bibliothèque écrite initialement en Haskell, permet d'écrire des tests basés sur la spécification des méthodes. Ainsi, l'exemple suivant vérifie que les méthodes Joiner.join et Splitter.split sont inverses l'une de l'autre :

```
1  import net.java.quickcheck.Generator;
2  import net.java.quickcheck.generator.iterable.Iterables;
3  import net.java.quickcheck.generator.PrimitiveGenerators;
4
5  @Test public void testSplitonStrings() {
6    for (List<String> words : someNonEmptyLists(strings())) {
7      String input = Joiner.join(words);
8      Iterable<String> letters = Splitter.split(input);
9      assertEquals(words, Lists.newArrayList(letters));
10   }}
```

Les fonctions telles que someNonEmptyLists() et strings(), sont de type Generator<T> et produisent un Iterable<T>. La méthode de test est exécutée sur chacun des éléments à l'intérieur de cet itérateur, en utilisant la syntaxe du **for-each** de Java.

Mettons à profit cette bibliothèque pour commencer à écrire des tests pour la classe Editor. Pour cela, il va être nécessaire de créer notre propre générateur, opération simplifiée par le fait que QuickCheck fournisse des générateurs primitifs, ainsi que des méthodes permettant de combiner ces générateurs (cf. `http://theyougen.blogspot.com/2009/07/generators.html`).

1. Écrire un générateur BufferGenerator qui implémente Generator<Buffer>, et cela en utilisant le générateur strings().
2. Écrire un test pour la méthode insert de la classe Buffer basé sur le générateur précédent, et un générateur d'entiers pour les positions d'insertion.
3. Écrire un test pour la classe Editor basé sur la spécification suivante :

   « Le contenu d'un Buffer doit rester inchangé si l'on exécute à la suite un killRegion() et un yank(), et cela quelle que soit la région capturée. »
4. Quels sont les avantages et les inconvénients du test par spécification, quand on les compare d'une part aux autres méthodes de test vues jusqu'ici, et d'autre part aux méthodes de vérification et de validation vues dans les cours d'IF300/IT304/IT305 ?