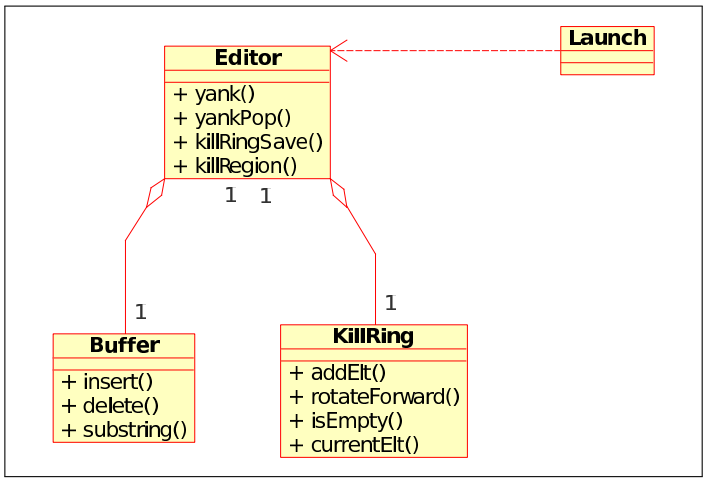
ENSEIRB
Année 2012-2013

3ème année
IT306

TD n°3 - Bouchons, faussaires et compagnie

Dans cette feuille, nous continuons avec l'architecture vue précédemment :



Pour commencer à faire du test d'intégration, on va vouloir tester la classe Editor seule. Or cette classe dépend de deux autres classes. Si le test doit être le plus local possible, il est nécessaire de rendre la classe Editor moins dépendante. Pour cela, on utilise des *bouchons*, aussi appelés *faussaires* ou *mock objects*. Un mock object est un objet qui se substitue à un autre objet et simule son comportement sans effectuer réellement de calculs. Il peut être utilisé pour tracer le comportement d'un objet.

Exemple : un mock object pour une base de données peut se contenter de renvoyer toujours le même élément constant lorsque l'on fait une requête sur un élément de la base. Cela évite ainsi de se connecter réellement à la base de données.

Exercice 1: Test par scénario

Pour implémenter les objets bouchons, une technique possible [1] consiste à construire des objets-enregistreurs, qui s'occupent de consigner l'ensemble des appels qui leur sont faits, puis de valider ces séquences d'appels.

1. Comment assurer qu'un mock object ait le même comportement que l'objet auquel il se substitue ?
2. Quelles sont les transformations nécessaires à appliquer au code de Editor pour pouvoir, dans la classe EditorTest, tester Editor avec des bouchons pour Buffer et KillRing ?
3. Appliquer ces transformations, et écrire deux classes BufferMock et KillRingMock qui ont un comportement minimal.

Notre problème ici consiste à tester le comportement de la classe Editor, et de son interaction avec les deux classes qu'il agrège. Pour cela, il est nécessaire de pouvoir *tracer* le comportement des mock objects.

1. Cette technique est utilisée entre autres par le framework EasyMock (cf. `http://easymock.org/`)

4. Ajouter dans chaque class Mock une variable String trace. Ensuite, faire que l'appel à chaque méthode du mock object écrive une trace dans cette variable. Par exemple :

   "addElt:currentElement:rotateForward:currentElement"

L'éditeur ne fait pas énormément de calculs par lui-même, et a tendance à déléguer la plupart des opérations à ses classes filles. Néanmoins, à un endroit du code, la classe assure que les paramètres donnés à la méthode substring sont dans le bon ordre.

5. Écrire une classe de test EditorTest, qui contient une méthode de test vérifiant que l'éditeur met le curseur et la marque dans le bon ordre avant de faire un *kill*.

   Le test doit utiliser les mock objects définis précédemment, et vérifier que les appels reçus par BufferMock sont corrects.

La majorité du comportement de Editor consiste à passer les bonnes chaînes de caractère à la bonne classe. Pour déterminer quels sont les comportements à tester, une méthode consiste à réaliser le diagramme de séquence d'une utilisation de cette classe.

6. Écrire un tel diagramme de séquence.
7. A partir du diagramme de séquence, écrire deux tests qui permettent de vérifier que la classe Editor transmet bien correctement les informations entre ses classes filles.

Exercice 2: Faussaires évolués

Le langage Java dispose d'un certain nombre de bibliothèques qui simplifient l'écriture des faussaires, et qui permettent de spécifier de manière précise leur comportement.

▷ Mockito est une bibliothèque Java qui simplifie l'écriture de mock objects (`http://code.google.com/p/mockito`), et permet de ne pas avoir à créer de nouvelles classes à la main.

```
1 import static org.mockito.Mockito.*;
2
3 Locomotive mock_loco = mock(Locomotive.class);
4
5 when(mock_loco.smokes()).thenReturn(false); // stubbing
6 mock_loco.load_fuel(anyInt());              // executing
7  verify (mock_loco).load_fuel(anyInt());    //  verifying
```

Il est ainsi possible de spécifier le comportement des faussaires, de les manipuler dans des fonctions, et de vérifier pour les appels qu'ils reçoivent leur ordre et leurs paramètres.

1. Écrire un test vérifiant le comportement de l'éditeur lors d'un appel à la méthode yankPop seul, avec des bouchons de la bibliothèque Mockito.

Un des intérêts des bibliothèques comme Mockito consiste à pouvoir facilement gérer la séquence des messages reçus par les mock objects : un objet de type InOrder va enregistrer l'ordre des messages reçus, et les appels à la méthode verify permettent de vérifier qu'ils sont appelés dans l'ordre spécifié.
Il est possible de se référer à la documentation de Mockito accessible à `http://docs.mockito.googlecode.com/hg/latest/org/mockito/Mockito.html`.

2. Écrire un test vérifiant que l'appel à killRegion se passe correctement en utilisant la bibliothèque Mockito. Pour cela, vérifier que le message envoyé par le Buffer est récupéré par le KillRing.

Pour continuer : à la suite d'une telle série de tests, nous avons une confiance suffisante dans le comportement de chacune des classes prise indépendamment des autres. En fait, il resterait encore à faire un test d'intégration en testant le comportement des classes réelles entre elles, sans mock objects.

Exercice 3: Analyse statique de code

L'analyse statique d'un programme est une analyse réalisée sans procéder à l'exécution du code source. Sous sa forme la plus simple, il s'agit de reconnaître des motifs dans le code connus pour contenir des fautes, qu'il s'agisse de simples erreurs, de mauvaises pratiques ou pire d'anomalies introduisant des failles dans le code. Parmi ces outils, on trouve FindBugs (cf. `http://findbugs.sourceforge.net`).

Considérons l'exemple suivant [2] :

```
1 class MutableDouble {
2     private double value_;
3     public boolean equals(final Object o) {
4       return (o instanceof MutableDouble) &&
5               (((MutableDouble) o).doubleValue() == doubleValue());
6       }
7     public Double doubleValue() {
8       return value_;
9     }}
```

1. Quel faille contient ce morceau de code ? En quoi est-il intéressant de laisser la détection de ce type de faille à un logiciel d'analyse statique ?

> ▷ La liste des vérifications réalisées par findbugs est accessible à l'adresse `http://findbugs.sourceforge.net/bugDescriptions.html`

2. Utiliser findbugs pour rechercher les vulnérabilités à l'intérieur du code de crystal.
   Quels types d'erreurs êtes-vous capable de détecter ?
3. Utiliser findbugs pour rechercher les vulnérabilités à l'intérieur de votre propre code.
   Quels types d'erreurs êtes-vous capable de détecter et de corriger ?
4. Quelles sont les avantages et les limites de ce type d'analyse de code ?

Un serveur d'évaluation de la qualité du code, Sonar (`http://www.sonarsource.org`), permet d'automatiser ces analyses statiques tout au long du développement.

2. Exemple tiré de la présentation *Mistakes that Matter*, de W. Pugh, un des auteurs du logiciel FindBugs. L'exemple lui-même est tiré de code d'un logiciel de Google.