1. Rappels généraux

N'oubliez pas de déposer votre travail sur Moodle à *chaque* séance. Vous pouvez le continuer chez vous après la séance et le re-déposer. Le travail doit rester entièrement personnel, mais vous pouvez demander de l'aide, conseils et informations.

2. Présentation générale

Mockito est un outil pour tester les interactions entre plusieurs classes : le fait que telle méthode appelle telle autre méthode, éventuellement dans une autre classe. Par exemple, on est en train de tester la méthode `m1` de l'objet `O1` et on veut savoir si `m1` appelle exactement n fois la méthode `m2` de l'objet `O2` avec les bons paramètres.

Il y a deux aspects dans Mockito :

- Un mode *espionnage* qui permet tracer les appels de méthode, c'est à dire savoir que lorsqu'on appelle telle méthode d'une classe, alors telle autre méthode est appelée, tant de fois et avec tels paramètres.
- Un mode *maquette* (*mocking*) qui consiste à faire croire à des tests unitaires qu'une classe existe pleinement, alors qu'elle n'est pas encore programmée ou que c'est seulement une interface.

Kaamelott Saison 3 - Épisode 95 - « L'étudiant » - Alexandre Astier

- Perceval : Eh ben. Vous allez me dire qu'un petit machin comme ça, ça jette des cailloux comme ça.
- Arthur (*désignant la catapulte*) : C'est une maquette, ça.
- Perceval : Une maquette ? Vous avez pas dit que c'était une catapulte ?
- Arthur : J'avais dit @Mock Catapulte maquette;

Nous allons employer Mockito pour vérifier la bonne mise en œuvre de patrons de conception dans différents projets. Un patron de conception définit des règles de fonctionnement d'une ou plusieurs classes, et on veut être certain(e) qu'elles ont été correctement programmées.

3. Mise en place du TP

3.1. Création du projet TP5

1. Créez un nouveau projet dans le *workspace* Eclipse :
   - GroupID : `fr.iutlan.ql`
   - ArtifactID : `tp5`
2. Téléchargez ce fichier ZIP . Il contient les sources et le fichier pom. Il faut entièrement remplacer le dossier `tp5` créé par Eclipse par celui du ZIP – c'est à dire dans le ZIP, il y a un `pom.xml` et un dossier `src` qui doivent remplacer ceux du projet que vous venez de créer.
3. Mettez à jour le projet Maven (menu `Project`, item `Update Maven Project`).

Dans la suite, vous allez utiliser plusieurs *packages*, chacun pour un patron de conception différent.

4. Test du patron *file d'attente*

On va commencer par quelque chose de simple. On veut vérifier le bon fonctionnement d'un patron « file d'attente » (*fifo queue* en anglais). Dans ce patron, il y a une instance de `FifoQueue` possédant essentiellement deux méthodes, `put(obj)` et `obj get()`. La première enregistre successivement des objets dans une mémoire interne du `FifoQueue`. La seconde récupère les objets un par un, dans le même ordre que l'enregistrement. Il y a une troisième méthode pour indiquer si la file est vide. C'est ce qu'on va vérifier.

☛ Dépliez les sources `main` et `test` du *package* `fr.iutlan.ql.tp5.fifo`, et regardez les fichiers qui s'y trouvent.

- Interface `FifoQueueSpec` : elle spécifie les méthodes à implémenter dans la classe `FifoQueue`.
- Classe `FifoQueue` : elle implémente l'interface.
- Classe `TestsFifoQueue` : les tests unitaires.

Il y a trois autres classes qu'on regardera plus tard :

- Class `Printer` : un exemple d'utilisation de `FifoQueue` sur des chaînes.
- Classe `TestsPrinter` : voir exercice 5 suivant.
- Classe `TestsPrinterWithMockFifoQueue` : voir exercice 6 suivant.

☛ Complétez les tests qui sont décrits dans les commentaires en utilisant AssertJ. Vous devrez ajouter d'autres tests pour détecter tous les problèmes.

Il y a un mécanisme dans `FifoQueue.java` pour introduire des bugs. La variable `bugs`, une liste, doit être vide pour que la file ait le comportement attendu, mais les valeurs `Bug.BUG*` introduisent des anomalies que vos tests devront détecter. Il y a des lignes commentées pour montrer des exemples d'affectation de cette variable et vous avez accès au source pour voir ce qui se passe (dans un vrai projet ou comme dans le TP3, ce ne serait pas forcément le cas). Vos tests doivent détecter chaque bug par au moins une *failure* ou une *error*. Et vos tests doivent tous passer quand il n'y a pas de bug.

Ce patron de conception a pu être testé sans faire appel à Mockito. C'est pour vous donner l'idée du TP. Les exercices suivants vont nécessiter Mockito pour isoler les vérifications.

5. Tests d'intégration

On s'intéresse à la classe `Printer`. Elle utilise une file d'attente pour stocker des lignes et pouvoir les imprimer ultérieurement. On lui fournit des lignes une par une avec la méthode `print`. Quand on a fourni toutes les lignes voulues, on appelle la méthode `flush` qui écrit toutes les lignes sur l'écran. Il y a un exemple d'utilisation avec sa méthode statique `main`.

On voudrait savoir si cette classe appelle correctement la file d'attente. Normalement, chaque ligne fournie à `print` conduit à un appel à `put` et `flush` doit ensuite faire autant d'appels à `get`. On va s'en assurer avec Mockito.

NB: cet exercice est un peu forcé car il y a très peu de différences entre `Printer` et la file d'attente. Il ne sert qu'à illustrer les concepts des tests d'intégration. Il faudrait imaginer une classe bien plus complexe avec d'autres structures de données connectées. Remarquez que dans la classe `Printer`, il y a un constructeur public et l'autre est limité au package. Seuls les tests peuvent l'appeler. La méthode `sysout` présente aussi cette caractéristique.

Nous allons créer une instance de `Printer` avec une file d'attente que nous allons espionner. Chaque appel de méthode sur cette file sera enregistré par Mockito, avec ses paramètres et on va faire des vérifications dessus.

👉 Enlevez tout bug de `FifoQueue`, sinon toute la suite va planter.

👉 Regardez `TestsPrinter.java`.

Voici quelques informations et explications pour comprendre. Le principe est d'appliquer l'annotation `@Spy` sur un objet. Il passe sous surveillance. Cet objet est ensuite utilisé comme on veut, mais en ajoutant l'annotation `@InjectMocks` aux autres objets qui l'utilisent. Regardez comment `printer` est créé dans `TestsPrinter.java`.

NB: pour faire ce genre de tests, il faut évidemment prévoir des méthodes pour pouvoir injecter des objets espionnés. Si la file était créée à l'intérieur de `Printer`, on n'aurait aucun moyen de la surveiller.

Après ces préparatifs, on arrive au plus intéressant : dans chaque test, Mockito mémorise tous les appels aux méthodes des objets espionnés et offre des assertions comme `verify(objet).méthode(paramètres...);` qui est vraie si la méthode a été appelée sur l'objet avec ces paramètres.

Voir aussi la documentation de Mockito . Il y a des variantes pour compter le nombre d'appels, voir la doc, § Verifying exact number of invocations :

- `verify(objet).meth(params...);` = `verify(objet, times(1)).meth(params...);`
- `verify(objet, times(N)).meth(params...);` vraie si cette méthode a été appelée exactement N fois.
- `verify(objet, atLeast(N)).meth(params...);` vraie si cette méthode a été appelée au moins N fois.
- `verify(objet, atMost(N)).meth(params...);` vraie si cette méthode a été appelée au maximum N fois.
- `verify(objet, never()).meth(params...);` vraie si cette méthode n'a jamais été appelée.

Et pour être certain qu'il n'y a eu aucune autre méthode appelée, on ajoute cette assertion :

- `verifyNoMoreInteractions(objet);` vraie s'il n'y a pas eu d'autres appels de méthodes sur l'objet que ceux mentionnés dans les `verify` précédents.

Comme toutes les interactions sont comptées, y compris celles de l'initialisation du test, il peut être utile de mettre les compteurs à zéro à la fin de la partie *ARRANGE* :

- `clearInvocations(objet);` efface l'historique des appels de méthodes sur les maquettes indiquées.

Dans certains cas, on n'a pas les valeurs précises des paramètres des méthodes appelées. On met alors une sorte de joker comme `anyInt()` ou `anyString()`. Par exemple, `verify(objet).meth(anyString());` Ce sont des instances de `ArgumentMatcher`, voir la doc . Il y a aussi `argThat` qu'on utilisera dans l'exercice 7.5.

Il y a encore un mécanisme à connaître dans Mockito. Il permet d'affirmer l'ordre des appels de méthode. On commence par créer une instance de `InOrder` qui regroupe tous les objets concernés, puis on vérifie les appels aux méthodes via cette instance.

👉 Dans `TestPrinter.java`, vous avez quelques exemples de tests en mode espionnage. Complétez-les pour mettre en évidence tous les bugs prévus dans `Printer`, c'est à dire qu'en activant un bug, au moins l'un de vos tests doit échouer (*failure* ou *error*), et ils doivent tous passer quand `bugs` est vide.

IMPORTANT Deux bugs vont vous échapper. La classe `Printer` appelle correctement la file d'attente, en ce qui concerne le nombre d'appels et les valeurs récupérées de la file, mais elle n'en fait pas ce qu'il faut. C'est pour ça qu'on va aussi surveiller la classe `Printer` elle-même.

👉 Pour attraper les deux bugs liés au comportement de `Printer`, ajoutez l'annotation `@Spy` sur l'instance de `printer` au début de `TestsPrinter.java`.

👉 Avec cette annotation, l'objet `printer` est lui-aussi sous surveillance. Complétez les tests, si nécessaire, avec des assertions sur le comportement de `printer`, ses appels à ses propres méthodes.

NB: C'est pour cela qu'il y a aussi des méthodes internes comme `sysout` dans la classe `Printer`, au lieu de faire des appels directs au système.

Dans les appels dans l'ordre, vous pouvez faire ce qui suit, car `InOrder` peut surveiller plusieurs objets ensemble :

```
InOrder inOrder = Mockito.inOrder(objet1, objet2);
inOrder.verify(objet2).methode2a("param1");
inOrder.verify(objet1).methode1a("param2");
inOrder.verify(objet2).methode2b("param3");
inOrder.verifyNoMoreInteractions();
```

Vous pouvez aussi définir plusieurs objets `InOrder` et tester les séquences séparément :

```
InOrder inOrder1 = Mockito.inOrder(objet1);
inOrder1.verify(objet1).methode1a("param1");
inOrder1.verify(objet1).methode1b("param3");
...
inOrder1.verifyNoMoreInteractions();

InOrder inOrder2 = Mockito.inOrder(objet2);
inOrder2.verify(objet2).methode2a("param2");
...
inOrder2.verifyNoMoreInteractions();
```

Vous pouvez donc élaborer des scénarios d'appels de méthodes et les vérifier.

IMPORTANT Ne partez pas dans des tests trop complexes, vous y passeriez trop de temps par rapport à l'objectif de mettre deux bugs en évidence.

6. Maquettage de la file d'attente

On monte d'un cran dans les tests d'intégration. On va se placer dans la situation où la classe `FifoQueue` n'a pas été programmée et on a quand même besoin de l'utiliser. En fait, on n'a que l'interface `FifoQueueSpec`, et une autre classe `Printer` qui veut l'utiliser.

Le principe consiste à faire une maquette de l'interface, c'est à dire une classe bidon qui donne

l'impression que l'interface est implémentée. La classe en question est bidon dans le sens où elle est uniquement capable d'apprendre des réponses par cœur, elle ne fait aucun calcul, aucun algorithme, rien d'autre que répéter ce qu'on lui a dit de dire. Elle simule ainsi la véritable classe mais uniquement sur des scénarios très précis d'appels à ses méthodes.

👉 Regardez `TestsPrinterWithMockFifoQueue.java`.

On a une variable annotée par `@Mock`. Elle n'est pas initialisée, car Mockito s'en charge en préparant un petit magnétophone prêt à enregistrer tout ce qu'on voudra.

La seconde variable est un `Printer` identique à celui de `TestsPrinter.java`. L'annotation `@InjectMocks` permet d'infiltrer cette variable avec la maquette, parce qu'on va espionner ce petit monde.

Alors il ne faut pas oublier que si `printer` appelle la moindre méthode de `file`, ça déclenchera une exception, du genre `MethodNotFound`, car `file` n'est pas une instance de `FifoQueue`, mais une « instanciation de l'interface » `FifoQueueSpec`, ce qui n'a aucun sens en Java.

C'est pour ça que dans les méthodes de test, il y a une phase pour préparer la file, lui faire apprendre ce qu'elle doit répondre quand on appelle telle méthode. Il y a deux façons de faire :

- `when(mock.méthode(paramètres...)).thenReturn(valeur);`
- `doNothing().when(mock).méthode(paramètres...);`

La seconde est à réserver aux méthodes qui retournent `void`. Elle est là pour ne pas avoir d'échec par méthode non trouvée.

Les paramètres sont des `ArgumentMatcher`, soit des constantes, soit des `anyString()`, `anyBoolean()`, etc.

Vous avez l'exemple de deux tests. Votre travail consiste à transposer les tests de `TestsPrinter.java` pour qu'ils marchent avec la maquette et qu'ils détectent les bugs de `Printer`.

7. Test du patron *État*

Le patron *État* ressemble beaucoup au patron *automate à états*. On a un objet qui peut être dans un état parmi plusieurs possibles et il y a des transitions entre les états, voir ces explications .

Ce patron ne peut pas être dissocié d'une application concrète. Ici, c'est une gestion de signalements de bugs, appelés « tickets ». Au début, un ticket est créé. Il peut être confirmé, commenté ou fermé. S'il est fermé avant d'être confirmé, alors il est refusé. Un ticket possède un titre et des commentaires optionnels.

7.1. Classes de l'application

👉 Dépliez les sources `main` et `test` du *package* `fr.iutlan.ql.tp5.state`, et regardez les fichiers qui s'y trouvent.

Voici un cheminement pour comprendre l'ensemble. Il faut distinguer deux choses, d'abord il y a la notion de ticket : un objet défini par un titre et des commentaires. Et il y a la notion d'état du

ticket, ouvert, confirmé, fermé, refusé. Ce sont les actions sur le ticket qui le font passer d'un état à l'autre. Voici un diagramme représentant les changements d'état en fonction des actions.

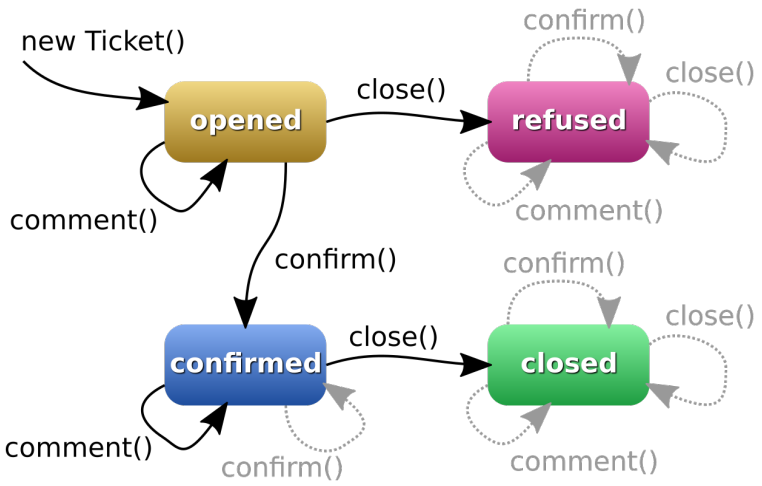

Figure 1: Automate à états

Les liens en gris clair sont les actions qui ne font rien.

Ces états sont représentés par des classes. Il y a d'abord celles qui représentent un ticket.

- La classe `Ticket` représente les détails d'un ticket, titre et liste de commentaires. C'est une sous-classe de `AbstractTicket`.
- La classe abstraite `AbstractTicket` définit la méthode `changeState` qui fait changer l'état du ticket.

👉 Il y a un bug d'entrée de jeu dans `AbstractTicket.java`, ligne 16. À vous de corriger...

On trouve ensuite les classes qui représentent l'état d'un ticket et ses changements.

- La classe abstraite `AbstractTicketState` mémorise un ticket et définit des actions par défaut sur ce ticket. Il y a un couplage bidirectionnel entre un ticket et son état : l'état est une variable membre du ticket (`AbstractTicket`) et le ticket est une variable membre de l'état.
- Il y a 4 classes concrètes dérivant de `AbstractTicketState` qui représentent l'état actuel d'un ticket : `TicketStateOpened`, `TicketStateConfirmed`, `TicketStateClosed`, `TicketStateRefused`. On passe d'un état à l'autre en faisant : `ticket.changeState(new TicketStateETAT());`.
- L'interface `TicketActions` spécifie toutes les actions qu'on peut faire sur un ticket. On peut appliquer ces actions sur un ticket, mais ça n'aura pas forcément un effet. Par exemple, rien ne se passe si on confirme un ticket fermé, et au contraire si on confirme un ticket juste créé, ça le change en confirmé. C'est dû au fait que les classes concrètes d'état ne surchargent pas toutes les actions.

On veut tester le fonctionnement du patron, la mécanique du changement d'état ainsi que les états et les actions qui les changent.

7.2. Test de la classe `AbstractTicket`

On commence par étudier la classe `TestsAbstractTicket`. Elle n'est pas terminée. Voici des explications pour finir le travail.

D'abord, `AbstractTicket` est une classe abstraite, donc on ne peut pas l'instancier. Et si on en faisait un mock, on ne pourrait pas vérifier ses méthodes. Ce sont les autres classes qu'on doit *mocker* pour vérifier les interactions de `AbstractTicket` avec ses satellites : des `TicketState*`, une par état possible.

Donc on définit une classe interne `AbstractTicketImpl`, dérivée de `AbstractTicket` mais quasi-vide pour surtout ne pas introduire de bugs à l'intérieur des tests.

☛ Les tests consistent à appeler chacune des méthodes de `AbstractTicket` en vérifiant ce qui se passe en interne ou avec les maquettes. L'un des tests est complet, inspirez-vous de lui pour les autres.

☛ Si vous n'avez pas corrigé le bug de `AbstractTicket.java`, l'un des tests va échouer.

7.3. Test de la classe `Ticket`

Cette classe de test, `TestsTicket`, est quasiment finie. Vous n'avez qu'une assertion à écrire. Mais il y a énormément de choses à comprendre. On va créer un objet `Ticket` très spécial : on va lui refiler des maquettes et des objets espionnés en guise de variables membres. Normalement, un ticket, en considérant toute sa hiérarchie est composé de :

- `String title;` : on le laisse tranquille, il est initialisé par le constructeur du ticket.
- `List<String> comments;` : on le remplace par une liste espionnée, ainsi toutes les modifications de cette liste seront enregistrées.
- `AbstractTicketState state;` : on le remplace par une maquette et cette maquette, on lui apprend à répondre à `toString()`.

Alors le problème, c'est qu'il n'y a pas de constructeur de `Ticket` qui prend toutes ces informations. Mais l'annotation `@InjectMocks` peut être utilisée ainsi :

```
@Mock Classe1 objet1;
@Spy Classe2 objet2 = ...;
@InjectMocks Classe3 objet3 = new Classe3(...);
```

Elle crée un vrai objet `objet3`, mais lorsque ses variables membres portent les mêmes noms que les *mocks* et *spy*, alors elles sont remplacées par ces *mocks*. Ici, on a donc fait exprès d'appeler les variables `state` et `comments` pour écraser les variables d'origine du ticket. Seul, le titre du ticket n'est pas remplacé.

NB: on aurait pu utiliser ça dans l'exercice sur la file d'attente.

Maintenant, vous pouvez comprendre comment fonctionnent ces tests. Il vous reste à coder la vérification de l'appel à `add` sur la liste des commentaires par `addComment`, et à vérifier avec AssertJ que la liste contient un et un seul élément, le commentaire ajouté.

7.4. Test de la classe `AbstractTicketState`

Ici aussi, on a une classe abstraite, donc on la dérive en `TicketStateCommonTest` sans rien définir de plus. Il suffit de maquetter un `Ticket` pour tester ce qu'on a besoin : le comportement des trois méthodes de l'interface `TicketActions`. Normalement dans cette superclasse abstraite, ces méthodes ne doivent rien faire. Elle représentent les flèches en gris dans la figure 1. Vous n'avez qu'à recopier les instructions d'un test à l'autre, mais en comprenant pourquoi.

7.5. Test de la classe `TicketStateOpened`

Le but dans `TestsTicketStateOpened` est de vérifier qu'il y a bien les changements d'états voulus pour un ticket ouvert. Étudiez attentivement l'appel à `verify`. Au lieu d'un paramètre normal pour `changeState`, on a `argThat(x -> { assertions sur x; return true })`, ce qui permet de tester le paramètre x avec toutes les assertions AssertJ. Mockito propose plusieurs mécanismes pour vérifier les paramètres des appels de méthodes, mais celui-ci est de loin le plus commode et le plus puissant.

Bien comprendre qu'un changement d'état s'effectue en appelant une méthode de l'interface `TicketActions` sur l'état du ticket. Cet état se fait remplacer par un autre.

👉 Terminez les tests.

7.6. Tests des autres classes `TicketState*`

Ils sont tous similaires à `TestsTicketStateOpened` et à terminer.

8. Rendu du TP

Vous devrez remettre votre travail à la fin de chaque séance. C'est ainsi que c'est noté, en contrôle continu.

Ouvrez un terminal bash dans le dossier de votre TP, là où il y a le fichier pom.xml, puis tapez ceci :

```
mvn clean
cd ..
tar cfvz tp5.tgz tp5
```

Puis déposez le fichier `tp5.tgz` dans la zone de dépôt du TP5 sur Moodle R4.02 Qualité de développement .

ATTENTION votre travail est personnel. Si vous copiez pour quelque raison que ce soit le travail d'un autre, vous serez tous les deux pénalisés par un zéro.

En cas de souci quelconque, envoyez un mail à `pierre.nerzic@univ-rennes1.fr` pour expliquer le problème.