1. Rappels généraux

Tous les TD et TP se dérouleront sur **Linux**.

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.

2. Présentation générale

Mockito est un outil pour tester les interactions entre plusieurs classes. 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 de savoir si lorsqu'on appelle telle méthode de l'une, 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 *Singleton*

On va commencer par quelque chose de simple. On veut vérifier le bon fonctionnement d'un patron « Singleton », voir ces explications.

1. Dépliez les sources `main` et `test` du *package* `fr.iutlan.ql.tp5.singleton`, et regardez les fichiers qui s'y trouvent.
   - Classe `Singleton` : sa méthode `getInstance` retourne le même objet à chaque appel.
   - Classe `TestsSingleton` : deux méthodes de test à compléter.
2. Complétez les tests qui sont décrits dans les commentaires en utilisant AssertJ.

Comme vous pouvez le constater, il y a des lignes commentées dans `Singleton.java`. C'est pour provoquer des erreurs qui devront être détectées par vos tests. Voici les manipulations :

a. Commentez la ligne `return instance;` et décommentez la ligne du `return null;`. Normalement le premier test ne doit plus passer.
b. Remettez les commentaires sur le `return null;` et décommentez le `return subinstance;`. Le premier test ne devrait pas passer. Il y a une assertion intéressante, `isExactlyInstanceOf` qui permet de faire une vérification plus stricte.
c. Remettez le commentaire sur le `return subinstance;` et maintenant, décommentez seulement le `return new Singleton();`. Le 2e test ne doit pas passer. C'est lui qui vérifie l'unicité du singleton, et il échoue si vous avez seulement comparé les deux variables par l'égalité, à cause de la méthode `equals` de la classe `Singleton`.
d. Enfin remettez tout comme c'était au début, et les deux tests doivent passer.

Ce patron de conception a pu être testé sans faire appel à Mockito. Les patrons suivants vont nécessiter Mockito pour isoler les vérifications.

5. Test du patron *Observateur*

On va travailler sur le patron « Observateur », voir ces explications.

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

- Interface `Subscriber` (*abonné*) : sa méthode `update` doit être appelée en cas de mise à jour par le `Publisher`.
- Classe `Publisher` : il gère une liste de `Subscriber` qu'il doit prévenir le cas échéant.

Il y a des lignes commentées, c'est pour provoquer des erreurs que vos tests devront détecter. On va faire passer des tests aux deux sources, en les séparant le plus possible.

5.1. Classe de test de `Publisher`

On commence par vérifier la classe `Publisher` avec la classe `TestsPublisher`. Voici quelques informations et explications pour comprendre. Voir aussi la documentation de Mockito.

5.1.1. Maquettes

Au début de `TestsPublisher`, on définit trois variables membres de type `Subscriber`, mais elles sont marquées de l'annotation `@Mock` parce que `Subscriber` n'est pas une classe mais une interface,

donc non-instanciable. Normalement, et ça sera le cas plus loin, on doit définir une classe qui implémente cette interface. Mais ici, on veut seulement faire des tests sur `Publisher` sans avoir une quelconque idée sur les instances de `Subscriber` qui seront fournies.

L'annotation `@Mock` sert, entre autres, à donner vie à une interface, à en faire des instances, sans avoir besoin de faire `new`. Alors par contre, ces instances sont creuses (*mock up* = maquette). Il ne se passera rien si on appelle leurs méthodes.

En attendant, avec ça, on peut faire des choses comme `publisher.addSubscriber(subscriber1);` et faire des vérifications côté `Publisher` comme si les abonnés étaient vrais.

En fait, même si `Subscriber` était une classe normale, le fait d'en faire des maquettes permet d'isoler totalement les tests sur `Publisher`. S'il y a des bugs côté `Subscriber`, ça n'aura aucun impact sur les tests de `Publisher`.

Une chose à savoir, c'est que l'annotation `@Mock` ne peut être mise que sur une variable membre, pas une variable locale. Ça oblige à partager les maquettes entre toutes les méthodes de test.

L'un des premiers tests de `TestsPublisher` n'est pas pleinement possible. À vous de deviner lequel. On ne peut que faire une vérification partielle. Ça vient du fait qu'il manque une méthode dans la classe `Publisher`, mais on ne la rajoutera pas, parce qu'elle correspond à un comportement totalement anormal (plusieurs inscriptions du même abonné). Vous pouvez essayer de vérifier dans un autre test, `addSeveralSameSubscriberMustBeOncePart2`, qu'un abonné qui est inscrit plusieurs fois, puis désabonné une seule fois, ne doit plus être abonné. Mais ça ne prouve pas totalement qu'on n'a pas des inscriptions multiples. Intéressant, non ?

5.1.2. Appels aux méthodes

Après des tests sur l'ajout et la suppression d'abonnés, on trouve des tests sur la méthode `updateSubscribers()` : le fait qu'elle doive appeler la méthode `update()` des abonnés. Ça devient intéressant. Mockito mémorise tous les appels aux méthodes des maquettes et offre des assertions comme `verify(maquette).méthode(paramètres...);` qui est vraie si la méthode a été appelée sur la maquette avec ces paramètres.

Il y a des variantes pour compter le nombre d'appels, voir la doc, § Verifying exact number of invocations :

- `verify(mock).meth(params...);` = `verify(mock, times(1)).meth(params...);`
- `verify(mock, times(N)).meth(params...);` vraie si cette méthode a été appelée exactement N fois.
- `verify(mock, atLeast(N)).meth(params...);` vraie si cette méthode a été appelée au moins N fois.
- `verify(mock, atMost(N)).meth(params...);` vraie si cette méthode a été appelée au maximum N fois.
- `verify(mock, 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(mocks...);` vraie s'il n'y a pas eu d'autres appels de méthodes sur ces *mock* 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 :

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

On verra plus loin que Mockito offre d'autres mécanismes pour filtrer les paramètres des méthodes qu'on surveille. Ici, ils ne sont pas nécessaires.

5.1.3. Ordre d'appels aux méthodes

On va ajouter une contrainte au patron observable, c'est que les abonnés soient prévenus dans l'ordre où ils se sont inscrits.

Le tout dernier test `updateMustUpdateAllSubscribersInOrder1` fait appel à un mécanisme de Mockito appelé `InOrder`. On commence par créer une instance de `InOrder` qui regroupe tous les *mocks* utilisés, puis on vérifie les appels aux méthodes via cette instance.

NB: l'instruction `InOrder inOrder = Mockito.inOrder(mock1, mock2, mock3, ...);` ne fait que rassembler les *mocks* concernés. Ce sont les instructions `inOrder.verify()` qui vérifient vraiment l'ordre des appels.

Alors il est possible, malheureusement que le test réussisse alors que la classe `Publisher` n'est pas correcte vis à vis de cette exigence. Ça vient de la classe `Collection` choisie pour stocker les abonnés. Le source `Publisher.java` laisse le choix entre :

- `LinkedList` : l'ordre des abonnés est préservé, mais un même abonné peut y être plusieurs fois
- `HashSet` : un abonné ne peut être présent qu'une seule fois, mais l'ordre d'inscription n'est pas conservé
- `LinkedHashSet` : unicité des abonnés et préservation de l'ordre d'inscription. C'est ce qu'il faut.

Il faut que vous écriviez deux méthodes de tests, quasiment identiques, mais avec un ordre d'insertion inversé, pour mettre la collection en défaut au moins une fois. Le pire serait une collection qui trie n'importe comment, mais par hasard toujours dans l'ordre attendu par le test au moment où on le fait... Ça peut arriver parce que ce sont les adresses des objets en mémoire qui sont prises comme critères de tri.

5.1.4. Travail à faire

1. Avec ces explications, vous devriez pouvoir finir de programmer les tests.
2. Comme pour le patron singleton, des lignes ont été commentées dans la classe testée. Le but des ces lignes est d'introduire des bugs dans `Publisher`. Par exemple, permettre plusieurs inscriptions distinctes du même `Subscriber`, ce qui ne doit pas être autorisé car ça conduit à faire plusieurs mises à jour du même élément. Un autre commentaire fait en sorte que les demandes d'inscription ne soient pas toutes acceptées et que les `Subscriber` ne soient pas tous mis à jour.

Faites en sorte que la classe de test détecte tous ces problèmes et n'accepte qu'un `Publisher` parfait.

5.2. Classe de test de `Subscriber`

Ça va être nettement plus restreint car ce n'est qu'une interface, mais ça va permettre de découvrir un autre aspect de Mockito. On va simuler la classe `Publisher` dans la classe `TestsSubscriber`.

Votre travail consiste à comprendre ce test, ce qu'il fait et ses limites, et compléter les assertions. Voici des explications.

D'abord, il y a une classe interne qui implémente l'interface `Subscriber`. On peut donc en faire des instances. Ça ne sera pas des maquettes mais de vrais objets.

Par contre, le `Publisher` est une maquette : un objet totalement vide, dont les méthodes ne font rien. Mais il y a une instruction, `when(mock.méthode(paramètres...)).thenReturn(valeur);`. Ça instruit la maquette de retourner une certaine valeur quand cette méthode avec ces paramètres est appelée. Il faut évidemment que la méthode retourne une valeur, pas `void`. Et l'autre contrainte, c'est que le mock ne soit pas `null`.

L'idée est que le `Publisher` est inerte, sauf quand on appelle très exactement la méthode `isSubscriber()` avec ce `subscriber`. Et dans ce cas-là, alors Mockito fait retourner `true` à la méthode.

Au cours du test, on fait différentes actions qui amènent à appeler le *getter* du `Publisher`. Suivez l'exécution : `isInscrit` appelle le *getter*. C'est un *mock*, mais Mockito a appris quelle valeur il doit retourner. Et donc l'assertion finale réussit. Commentez la ligne `when().thenReturn()` pour voir.

Malheureusement, ça ne marche que pour les méthodes qui retournent une valeur, et dans la classe `Publisher`, il n'y en a qu'une. Donc on ne peut pas faire davantage de tests. Ici, ce qui est vérifié seulement, c'est qu'il y a bien un appel à `publisher.isSubscriber(subscriber)` et qu'il retourne `true`, et comme on sait que la classe `Publisher` est vérifiée par ailleurs, on est sûr que tout est bon.

Toutefois, mémorisez le principe : la classe `Publisher` est une maquette. Elle possède un *getter* qui est susceptible d'être appelé par la classe à tester. On peut lui faire apprendre quoi retourner selon les appels et voir si la classe testée fait ce qui est prévu avec cette valeur. Ce concept sera utilisé plus loin.

Il y a une variante pour déclencher une exception :

```
when(mock.méthode(paramètres...)).thenThrow(new Exception());
```

Vous pouvez préciser la classe d'exception et ses paramètres.

NB: si vous définissez une directive `when(...).then...(...);` qui n'est pas utilisée, vous aurez une exception : « Unnecessary stubbings detected. ». On vous dit quelle ligne, par exemple `(TestsRequestHandler.java:50)` et on vous dit de l'enlever parce qu'elle ne sert à rien.

Le second test est un peu redondant par rapport à ce qui est essayé dans `TestsPublisher`. On veut vérifier qu'une mise à jour d'un abonné fait un changement dans l'abonné.

6. Test du patron *Chaîne de responsabilité*

On arrive à un patron appelé « Chaîne de responsabilités », voir ces explications.

Dépliez les sources `main` du *package* `fr.iutlan.ql.tp5.chainresp`. Dans la situation du TP, il y a trois sortes d'objets :

- Des **requêtes** représentées par l'interface `IRequest`. Une requête possède un type. Ce type est un `Enum` et fourni de manière générique.
- Des **exécuteurs de requêtes**, interface `IRequestHandler`. Un exécuteur est spécialisé dans certains types de requêtes. Il ne peut traiter que ces requêtes et une seule à la fois. Dans ce TP, une fois qu'il a sa requête à traiter, il la garde définitivement et ne peut plus traiter une autre requête.
- Un **gestionnaire d'exécuteurs** qui émet des requêtes, interface `IRequestEmitter`. Les exécuteurs sont chaînés dans une liste, et quand il y a une requête, le gestionnaire cherche le premier exécuteur libre et capable de traiter la requête. Il lui attribue la requête.

Ces interfaces sont implémentées dans un premier niveau :

- La classe `Request` implémente `IRequest`. En résumé, elle définit un constructeur et donne vie aux méthodes de l'interface.
- La classe abstraite `RequestHandler` implémente partiellement `IRequestHandler`. Il y a un peu plus de travail, car les méthodes font quelques vérifications. Ces vérification peuvent être désactivées par les variables `static final boolean CHECK_*`. C'est pour fragiliser volontairement cette classe afin de vérifier les tests unitaires. D'autre part, il y a une méthode abstraite, `canHandleRequest` car ce sont les implémentations finales qui savent quels requêtes elles peuvent traiter.
- La classe `RequestEmitter` implémente `IRequestEmitter`. Son travail consiste à choisir le premier `IRequestHandler` qui est à la fois libre et capable de traiter la requête, et de lui confier cette requête.

Il y a un second niveau, en dessous, avec un exemple d'utilisation de ce patron[1], toutes les classes `Orc*` du sous-package *example*. Lancez la méthode `main` de la classe `OrcKing`. Ce roi des Orcs possède plusieurs subordonnés, des `RequestHandler`, capables d'exécuter certaines tâches.

6.1. Classe de test de `RequestHandler`

On commence par vérifier la classe `RequestHandler` avec la classe `TestsRequestHandler`. Voici plusieurs tests à concevoir et réaliser. Chacun vérifie des règles tirées du patron de conception.

Pour vérifier ces règles, on se place dans un cadre spécifique, pas celui des Orcs mais avec des implémentations locales. Regardez la méthode `mkRequestHandler`. Elle instancie un `RequestHandler` en définissant uniquement la méthode abstraite `canHandleRequest`.

6.1.1. Test `chaineHandlersCompleteEtOrdonnee`

- Règle : la chaîne des `RequestHandler` doit être complète et dans le bon ordre.

Dans la partie *arrange*, il y a la définition de quatre `RequestHandler`. Ils sont ensuite liés l'un avec l'autre dans la partie *act*. Il vous reste à vérifier que le deuxième est bien le suivant du premier, que le troisième est celui du deuxième, etc. Attention, il faut comparer des objets directement,

[1] tiré de https://java-design-patterns.com/patterns/chain-of-responsibility/

et non pas avec `isEqualTo`, parce qu'il faut que ça soient les mêmes objets et pas d'éventuelles copies.

Comment pourrait-on faire échouer ce test ? Par exemple, en commentant la ligne `this.nextHandler = next;` qui est dans la méthode `setNextHandler`. Le `RequestHandler` deviendrait incapable de mémoriser le gestionnaire suivant, et donc la chaîne serait perdue.

6.1.2. Tests `detectHandlerCycle1` et `detectHandlerCycle2`

- Règle : la chaîne des `RequestHandler` ne doit pas contenir de cycle. C'est à dire que le suivant d'un `RequestHandler` ne doit jamais être l'un des `RequestHandler` qu'on a indiqué auparavant. La liste des `RequestHandler` doit être linéaire et se terminer par un `null`.

Votre travail consiste à compléter la définition des suivants dans la partie *act*, et à vérifier qu'il y a bien une exception émise dans ces deux tests. NB: l'exception est levée dans la méthode `setNextHandler`, vous devez seulement vérifier que c'est bien le cas.

Comment pourrait-on faire échouer ce test ? Par exemple, en mettant `CHECK_CYCLES` à `false` dans la classe `RequestHandler`, ou en ne créant pas de cycle dans le test.

6.1.3. Test `handlerLibreEtCapableDoitAccepterRequete`

- Règle : un `RequestHandler` qui est libre et capable de traiter une requête doit l'accepter.

On veut cerner au plus près ce que fait le `RequestHandler rh`. S'il est correctement programmé, alors il ne doit faire que demander à la requête de quel type elle est, puisqu'il est libre initialement. Donc on va utiliser une maquette de requête et lui faire apprendre ce qu'elle doit répondre. Voici les instructions concernées :

```
// crée une maquette de requête
@Mock IRequest<TestReqTypes4> mr1;

// mr1 doit répondre qu'elle est du type RQT1
when(mr1.getRequestType()).thenReturn(TestReqTypes4.RQT1);
```

Cette variable membre `mr1` est une instance d'interface. C'est impossible normalement, mais Mockito peut faire cela, il en fait un *mock*. C'est objet très particulier qui possède toutes les méthodes de l'interface, mais ces méthodes ne font rien du tout et retournent toutes `null`. La méthode `when` de Mockito fait apprendre à cet objet ce qu'il doit répondre quand on appelle l'une de ses méthodes, ici `getRequestType`.

Donc, ici, si jamais « quelqu'un » appelle `mr1.getRequestType()`, il recevra `TestReqTypes4.RQT1`.

La partie *act* du test effectue les mêmes actions qu'un `RequestEmitter`, vérifier que le *handler* peut traiter la requête et la lui assigner.

Vous devez compléter la partie *assert* pour finir le test.

Comment pourrait-on faire échouer ce test ? Par exemple, en commentant la ligne `this.handledRequest = request;` dans la méthode `RequestHandler.handle`. Ça le rendrait incapable de mémoriser la requête qu'il est censé traiter.

6.1.4. Test `handlerOccupeNePeutPasAccepterAutreRequete`

- Règle : un `RequestHandler` qui est capable de traiter une requête mais qui n'est pas libre, ne doit pas l'accepter.

Dans ce test, on crée un `RequestHandler rh` et on lui confie la requête `mr1`. Cette requête est une maquette et aucune de ses méthodes n'est appelée. On a une seconde requête `mr2` qui pourrait être prise en charge par `rh`.

Ensuite, dans la partie *act*, on demande à `rh` de traiter `mr2`. La partie *assert* doit vérifier que ça n'a pas été accepté.

Comment pourrait-on faire échouer ce test ? Par exemple, en mettant`CHECK_UNEMPLOYED` à false, ce qui fait accepter toute requête, même quand il y en a déjà une.

6.1.5. Test `auMoinsUnDesHandlersDoitTraiterCeType`

- Règle : dans la liste des exécuteurs, il doit y en avoir au moins un pour traiter chaque type de requête.

C'est un test qu'on doit effectuer sur l'application finale, les orcs. Ça n'a aucun intérêt de la faire sur les classes abstraites `RequestHandler` et `RequestEmitter`. Et d'autre part, il faut que ça soit un test paramétré, parce qu'il faut pouvoir s'adapter à un nombre variable de types de requêtes.

Vous devez seulement mettre en place le fait que le test est paramétré et comprendre comment ce test est effectué.

6.1.6. Autres tests

Voyez-vous d'autres règles de bon fonctionnement concernant les `RequestHandler` ? Évidemment, on pourrait tester le *getter* qui retourne le nom, mais c'est d'un intérêt moindre par rapport au patron de conception.

Pensez-vous qu'il soit acceptable de pouvoir appeler `RequestHandler.handle(null)` ? Que proposez-vous ? Programmez-le, à la fois dans le test et dans la méthode `RequestHandler.handle`.

6.2. Classe de test de `TestsRequestEmitter`

Maintenant, on va vérifier la classe `RequestEmitter` avec la classe `TestsRequestEmitter`. Les tests qu'on va faire sont des sortes de scénarii assez complexes qui effectuent plusieurs préparatifs avant de pouvoir agir, et ensuite il y a aussi de nombreuses vérifications.

6.2.1. Test `verifierRequestEmitterAvecUnSeulHandlerOk`

- Règle : le `RequestEmitter` doit interroger le premier `RequestHandler` et lui assigner la requête s'il est disponible et capable de la traiter.

Ce premier test montre comment créer un scénario pour de tels tests unitaires. La partie *arrange* met en place un gestionnaire avec une maquette de `RequestHandler`. On prépare les réponses que donnera cette maquette : libre et capable de traiter la requête.

La partie *assert* vérifie quatre choses : 1) le `RequestEmitter` a vérifié que le `RequestHandler` était libre, 2) qu'il est capable de traiter la requête, 3) le `RequestEmitter` lui a assigné la requête et 4) le `RequestEmitter` n'a pas cherché d'autres `RequestHandler`.

Votre seul travail sur ce test consiste à le comprendre entièrement.

6.2.2. Test `verifierRequestEmitterAvecUnSeulHandlerOccupe`

- Règle : le `RequestEmitter` doit interroger le premier `RequestHandler` et ne pas lui assigner la requête s'il n'est pas libre.

Complétez les parties *arrange* et *assert* selon les commentaires.

6.2.3. Test `verifierRequestEmitterAvecUnSeulHandlerPasCapable`

- Règle : le `RequestEmitter` doit interroger le premier `RequestHandler` et ne pas lui assigner la requête s'il n'est pas capable.

Complétez les parties *arrange* et *assert* selon les commentaires.

6.2.4. Test `verifierRequestEmitterAvecDeuxHandlersSecondOk`

- Règle : après avoir échoué sur un premier `RequestHandler`, le `RequestEmitter` doit interroger le `RequestHandler` suivant, et lui assigner la requête s'il est disponible et capable de la traiter.

Complétez les parties *arrange* et *assert* selon les commentaires.

6.2.5. Test `verifierRequestEmitterAvecDeuxHandlersAucunOk`

- Règle : après avoir échoué sur tous les `RequestHandler`, le `RequestEmitter` doit retourner `false`.

Complétez les parties *arrange* et *assert* selon les commentaires.

7. 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.