Sujet de l'EI 2021-2022

Conditions de l'épreuve	
Durée :	1h30
Documents autorisés :	Aucun document personnel n'est autorisé. Les supports de cours et les sujets des TP sont disponibles directement sur le poste de travail.

0. Préliminaires

Mise en place

- Créer un dossier qui servira de workspace Eclipse pour cette EI
- Télécharger le fichier **ZIP Outils-Java-EI-Revision.zip**
- Extraire son contenu dans le dossier workspace.
- Démarrer Eclipse et ouvrir le dossier workspace.

Clean général

- Pour être sûr d'avoir un environnement de travail propre, effectuez l'opération **clean** sur l'ensemble du workspace :
  - Menu "Project > Clean…".
  - Sélectionnez la case "Clean all projects", puis appuyez sur le bouton "Clean".

Structure de l'EI

Cette EI est découpée en 4 parties :

1. API Reflection	(4,5 points)
2. Spring	(7,5 points)
3. Maven	(2,5 points)
4. JUnit	(5,5 points)

Les 4 parties sont indépendantes et peuvent être traitées dans n'importe quel ordre.

1. API Reflection

Présentation

Le projet **exo-1-reflection**, contient 3 classes.

- La classe **Analyseur** est celle dont vous allez devoir écrire le code.
  Le but est d'avoir une classe capable d'analyser le code d'une autre classe grâce à l'API Reflection.
- La classe **Personne** est la classe qui est destinée à être analysée par l'**Analyseur**.
- **TestAnalyseur** est une classe de test, basée sur JUnit. Elle vous permettra d'exécuter les méthodes de la classe **Analyseur** et de savoir si votre code fonctionne correctement.

Exécution de la classe de test

- Faites le nécessaire pour que la classe **TestAnalyseur** soit exécutée.
  La barre doit être verte. Tout semble OK.
- Dans la vue "Junit", si vous dépliez la classe **TestAnalyseur**, vous devez voir qu'aucun test n'a fonctionné. En fait, ils sont tous désactivés (skipped).
- Ouvrez le code de la classe **TestAnalyseur**.
- Recherchez la 1ère méthode de test : celle qui est annotée **@Order(1)**.
- Supprimez l'annotation **@Disabled** qui est appliquée à cette méthode.
  Puis, ré-exécutez les tests.
  La barre doit être rouge. Le 1er test a échoué. Il concerne la méthode **listerVariables()** de la classe **Analyseur**.

Méthode listerVariables()

- Ouvrez le code de la classe **Analyseur** et recherchez la méthode **listerVariables()**.

Cette méthode a pour but de construire une liste de chaînes de caractères contenant les noms des variables de la classe analysée. Cette liste doit contenir toutes les variables de la classe analysée, y compris ses variables privées.

La classe analysée est celle qui est représentée par la variable **type** qui est passée en paramètre à la méthode.

- Écrivez le code qui permet de parcourir la liste de toutes les variables de la classe analysée (y compris celles qui sont privées). Ce code doit ajouter à la liste, le nom (**name**) de chacune de ces variables.
- Ensuite, exécutez la classe de test.
  Vous devez constater que le premier test a réussi et qu'il est passé en vert.

Dans la vue "Console" d'Eclipse, vous devez avoir l'affichage ci-dessous. Il s'agit de la liste des variables de la classe **Personne**.

```
Liste des variables (5)
-----------------------
nom
prenom
dateNaissance
age
majeur
```

Méthode listerMethodes()

- Dans la classe **TestAnalyseur**, supprimez l'annotation **@Disabled** de la méthode de test n°2.
  À présent, si vous exécutez la classe de test, le test n°2 va échouer.
- Dans la classe **Analyseur**, recherchez la méthode **listerMethodes()**.

Cette méthode doit faire la même chose que la précédente, sauf qu'elle concerne les méthodes et non plus les variables de la classe analysée.

- Écrivez le code qui permet de parcourir la liste de toutes les méthodes de la classe analysée (y compris celles qui sont privées), et qui ajoute à la liste le nom de chacune de ces méthodes.
- Ensuite, exécutez la classe de test. Le test n°2 doit réussir.
  La vue "Console" doit contenir l'affichage suivant :

```
Liste des méthodes (8)
----------------------
toString
getAge
isMajeur
getDateNaissance
calculerAge
actualiserAge
getNom
getPrenom
```

Méthode listerMethodesObsoletes()

- Dans la classe **TestAnalyseur**, supprimez l'annotation **@Disabled** du test n°3.
- Dans la classe **Analyseur**, recherchez la méthode **listerMethodesObsoletes()**.

Le code à écrire doit faire presque la même chose que celui que vous avez écrit précédemment. La différence est que seules les méthodes obsolètes (c'est-à-dire celles qui ont une annotation **@Deprecated**) doivent être ajoutées à la liste.

- Vous pouvez procéder par copier-coller à partir du code de **listerMethodes()**.
  Mais, vous devez faire en sorte que seules les méthodes annotées avec **@Deprecated** aient leur nom ajouté à la liste.
- Ensuite, exécutez la classe de test. Le test n°3 doit réussir.
  La vue "Console" doit contenir l'affichage suivant :

```
Méthodes obsolètes (2)
----------------------
isMajeur
actualiserAge
```

Méthode instancier()

- Dans la classe **TestAnalyseur**, supprimez l'annotation **@Disabled** du test n°4.
- Dans la classe **Analyseur**, recherchez la méthode **instancier()**.

Cette méthode a pour but de créer un objet de la classe analysée et de le retourner.

- Écrivez le code qui permet de créer un nouvel objet de la classe représentée par la variable **type**. Puis, retournez cet objet.
- Ensuite, exécutez la classe de test. Le test n°4 doit réussir.
  La vue "Console" doit contenir l'affichage suivant :

```
Instanciation
-------------
null null null 0 false
```

Méthode executerMethode()

- Dans la classe **TestAnalyseur**, supprimez l'annotation **@Disabled** du test n°5.
- Dans la classe **Analyseur**, recherchez la méthode **executerMethode()**.

Cette méthode reçoit en paramètre un objet **bean** de type **T**, le nom d'une méthode et une valeur.

Le traitement à effectuer consiste à rechercher dans l'objet **bean**, une méthode qui correspond au nom indiqué et qui accepte un paramètre de même type que l'objet **valeur**. Ensuite, il faut exécuter cette méthode en lui passant la **valeur**.

- Écrivez le code qui permet de récupérer dans une variable, un objet qui représente la méthode recherchée au sein de la classe de l'objet **bean**. Pour trouver cette méthode, il y a 2 critères : son nom et le type de son paramètre (qui est le type de l'objet **valeur**).
- Ensuite, il faut faire le nécessaire pour que la méthode ainsi récupérée soit accessible, même si elle est privée.
- Enfin, demandez l'exécution de cette méthode en l'appliquant à l'objet **bean** et en lui passant la **valeur** à traiter.

- Le test n°5 doit réussir.
  La vue "Console" doit contenir l'affichage suivant :

```
Exécution méthode
-----------------
DUPONT Jean 06/12/1997 24 true
```

Méthode affecterVariable()

- Dans la classe **TestAnalyseur**, supprimez l'annotation **@Disabled** du test n°6.
- Dans la classe **Analyseur**, recherchez la méthode **affecterVariable()**.

Cette méthode reçoit en paramètre un objet **bean** de type **T**, le nom d'une variable et une valeur.

Le traitement à effectuer consiste à rechercher dans l'objet **bean**, une variable qui a le même nom que celui passé en paramètre. Pus, il faut stocker la **valeur** dans cette variable.

- En vous inspirant de ce que vous avez fait à l'étape précédente, écrivez le code qui permet de récupérer un objet qui représente la variable recherchée au sein de la classe de l'objet **bean**.
- Ensuite, il faut faire le nécessaire pour que la variable ainsi récupérée soit accessible, même si elle est privée.
- Enfin, faites ce qu'il faut pour affecter à cette variable, au sein de l'objet **bean**, la **valeur** reçue en paramètre
- Le test n°6 doit réussir.
  La vue "Console" doit contenir l'affichage suivant :

```
Affectation valeur
------------------
DUPONT Jean 17/05/2000 99 true
```

2. Spring

Le projet **exo-2-spring**, contient une version de l'application LaboFX.

Dans son état actuel, cette version n'utilise pas Spring, mais une classe **Context** artisanale créée par le développeur de l'application.

Dans un premier temps vous allez commencer par tâcher de faire fonctionner cette application en utilisant cette classe **Context**.

Classe AppliLaboFX

- Ouvrez le code de la classe **AppliLaboFX**.
  Ce code est inachevé. Il faut le compléter.
- Ligne 24 : la variable **managerGui** est initialisée à null. C'est une erreur.
  Il faut utiliser l'objet **context** pour l'initialiser.
- Ligne 28 : même problème avec la variable **modelCalcul**.
  De nouveau, utilisez l'objet **context** pour l'initialiser.
- Ligne 37 : toujours la même chose … et la même solution.

Injections de dépendances

- Démarrez l'application en exécutant la classe **AppliLaboFX**.
  Cela génère des erreurs, car le développeur a omis de déclarer les injections de dépendances.
- Recherchez la cause de l'erreur. Il s'agit d'une **NullPointerException**: qui s'est produite dans la classe **ManagerGui**. C'est la variable **context** qui vaut null, car elle n'a pas été initialisée.
  Faites le nécessaire pour qu'elle soit initialisée par injection de dépendance.
- Exécutez de nouveau l'application. Vous allez encore avoir une **NullPointerException**.
  Résolvez le problème et recommencez autant de fois que nécessaire, jusqu'à ce qu'il n'y ait plus d'erreurs. Y compris lorsqu'on actionne le bouton "Nouv. Fen.".

Méthode d'initialisation

- Ouvrez le code de la classe **AppliLaboFX**.
- Mettez en commentaires, les 2 dernières instructions de la méthode **start()**. C'est-à-dire :

```
ModelCalcul modelCalcul = ... ...;
modelCalcul.init();
```

Ces 2 instructions avaient pour but d'appeler la méthode **init()** du **ModelCalcul**.
Normalement, il devrait être possible d'indiquer au **Context** qu'une méthode est une méthode d'initialisation afin qu'il l'exécute automatiquement après l'instanciation du composant.

- Ouvrez le code de la classe **ModelCalcul**.

- Faites le nécessaire pour indiquer que la méthode **init()** est une méthode d'initialisation qui doit être exécutée automatiquement par le **Context**.
- Démarrez l'application.
  La vue "Console" doit contenir la ligne :
  labofx.model.DaoCalculSerial#lire()
  Cela prouve que la méthode **init()** a bien été exécutée.

Méthode de clôture

- Dans la classe **ModelCalcul**, faites le nécessaire pour indiquer que la méthode **close()** est une méthode de clôture qui doit être appelée juste avant l'arrêt de l'application.
- Dans la classe **AppliLaboFX**, allez dans la méthode **stop()**.
- Mettez en commentaire les 2 instructions qui s'y trouvent.
- Puis, à leur place, écrivez l'instruction qui indique au **context** que l'application est en train de s'arrêter et qu'il doit exécuter la procédure de fermeture.
- Démarrez l'application, puis arrêtez-la.
  La vue "Console" doit contenir la ligne :
  labofx.model.DaoCalculSerial#enregistrer()
  Cela prouve que la méthode **close()** du **ModelCalcul** a bien été exécutée.

Utilisation de Spring

À présent que l'application fonctionne, il est temps de passer à Spring.

Au préalable, il faut indiquer à Eclipse l'emplacement des différentes bibliothèques fournies par le framework.

Pour cela, une User Library nommée **Spring** a déjà été créée. Il suffit de l'ajouter au projet.

- Ajoutez au projet **exo-2-spring**, la User Library **Spring**.
- Dans le package **labofx**, supprimez la classe **Context**, vous n'en avez plus besoin.

Des erreurs sont apparues. Vous allez les corriger dans les étapes suivantes.

Création de la classe de configuration de Spring

- Dans le package **labofx**, créez une nouvelle classe nommée **ConfigLaboFX**.
- Ouvrez le code de la classe **ConfigLaboFX**.
- Appliquez à la classe, l'annotation qui indique à Spring que l'on veut utiliser le mécanisme de scan de composants.
- Dotez cette annotation d'un attribut qui indique que l'on veut activer le mode Lazy.
- Enregistrez les modifications.

Adaptation de la classe AppliLaboFX

- Ouvrez le code de la classe **AppliLaboFX**.
- Faites le nécessaire pour que la variable **context** ne soit plus de l'ancien type **Context**, mais du type fourni par Spring qui peut être configuré grâce aux annotations. Le nom de cette classe commence par **AnnotationConfig**.
- Dans la méthode **start()**, supprimez l'instruction qui est en erreur, et remplacez-la par les 3 instructions qui sont nécessaires pour initialiser un contexte Spring :

Créer l'objet contexte Spring et stocker sa référence dans la variable **context**.

1. Enregistrer dans ce contexte, la classe de configuration **ConfigLaboFX**.

Exécuter la méthode **resfresh()** du contexte.

- Enregistrez les modifications que vous venez d'apporter à la classe **AppliLaboFX**.
  À ce stade, il ne devrait plus y avoir aucune erreur dans cette classe.
  S'il y en a, faites le nécessaire pour les faire disparaître.

Adaptation de la classe ManagerGui

- Ouvrez le code de la classe **ManagerGui**.
- Le type de la variable **context** est inadapté. Remplacez-le par un type approprié.
  Plusieurs solutions sont possibles. Ici, l'idéal est d'indiquer l'interface dont héritent tous les contextes utilisés par Spring. Si vous la connaissez, utilisez-la. Sinon, faites en sorte que l'erreur disparaisse en utilisant un type compatible.
- Enregistrez la modification de la classe **ManagerGui**.
  Normalement, il ne doit plus y avoir d'erreurs.
  Sinon, faites le nécessaire pour les faire disparaître.

Identification des composants à instancier

Dans la classe de configuration, on a choisi d'utiliser le mécanisme d'exploration de packages (scan de composants) pour identifier les classes qui peuvent être instanciées par Spring.

Lorsqu'il explore les packages, Spring recherche des classes qui sont caractérisées par la présence d'une certaine annotation.

- Ajoutez cette annotation à toutes les classes qui en ont besoin.
  Attention ! Une pénalité sera attribuée si l'annotation est mise à des classes qui n'en ont pas besoin.
  NB : La classe **ConfigLaboFX** ne doit pas être modifiée. Le seul travail à faire consiste à ajouter une annotation dans certaines classes de l'application.

Voici une technique pour vérifier que vous avez bien fait ce qu'il fallait :

- Démarrez l'application en exécutant la classe **AppliLaboFX**.
  Tout doit fonctionner sans aucune erreur.
- Dans la vue "Console" d'Eclipse, observez les classes qui ont été instanciées.

Il y en a un certain nombre. Cela doit correspondre exactement au nombre d'annotations que vous avez ajoutées.

Si vous avez ajouté des annotations dans des classes qui ne sont pas listées dans la vue "Console", c'est qu'elles sont inutiles. Supprimez-les.

Instanciation du contrôleur

Notre application a un défaut. Si on actionne le bouton "Nouv. Fen.", cela permet d'ouvrir plusieurs fenêtres, mais elles utilisent toutes le même contrôleur.

En effet, dans la vue "Console", il y a une seule ligne :

new labofx.gui.ControllerCalcul

Cela montre qu'un seul objet est créé par Spring.

Or, ceci n'est pas conforme aux préconisations de JavaFX. Normalement, chaque fenêtre doit être associé à un objet contrôleur différent.

- Faites le nécessaire pour que chaque fois que l'on demande à Spring de fournir la référence d'un objet contrôleur, celui-ci crée une nouvelle instance de la classe **ControllerCalcul**.
- Exécutez l'application. Chaque fois que l'on ouvre une nouvelle fenêtre, il doit y avoir dans la vue "Console", une nouvelle ligne qui est affichée :

  new labofx.gui.ControllerCalcul

DaoCalculXml

Dans cette partie, il vous est demandé d'ajouter un composant supplémentaire à l'application : un nouveau DAO qui permet d'utiliser un fichier au format XML.

Récupération de la classe DaoCalculXml

- Le projet **exo-2-spring**, contient un dossier **dao**.

  Dans ce dossier, récupérez la classe **DaoCalculXml** et placez-la dans le package **labofx.model**.

Votre application est désormais composée de 2 DAOs qui sont interchangeables. Il faut utiliser soit l'un, soit l'autre.

Actuellement, c'est le DAO Serial qui est utilisé. Vous allez faire le nécessaire pour que l'application utilise le DAO XML.

Adaptation de la classe ModelCalcul

- Ouvrez le code de la classe **ModelCalcul**.

  Vous constatez que la variable **daoCalcul** est de type **DaoCalculSerial**.
- Faites le nécessaire pour que le **ModelCalcul** puisse utiliser indifféremment n'importe lequel des 2 DAOs sachant qu'ils héritent tous les deux d'une même interface.

  Il suffit de changer le type de la variable **daoCalcul**.

Modification de la classe de configuration

- Dans le package **labofx**, ouvrez le code de la classe de configuration de Spring.
- Ajoutez-lui une méthode qui indique à Spring comment instancier le composant qui hérite de l'interface **IDaoCalcul**.

  Il faut écrire une méthode ayant le prototype suivant :

  ```
  public IDaoCalcul daoCalcul()
  ```

  Cette méthode doit être configurée de telle façon que ce soit la classe **DaoCalculXml** qui soit instanciée.

  N'oubliez pas l'annotation qui est nécessaire pour que Spring considère cette méthode comme une définition de composant à instancier.
- Faites aussi le nécessaire pour que l'instanciation de ce composant ait lieu en mode Lazy.

Test du fonctionnement

- Démarrez l'application en exécutant la classe **AppliLaboFX**.

  Vous devez constater que le DAO qui est utilisé est bien le DAO XML.

3. Maven

Le projet **exo-3-maven** est un projet Java qui n'utilise pas Maven (pas encore)

Il s'agit d'un projet Eclipse tout à fait standard, mais on a organisé le code-source selon les standards définis par Maven.

- On a réparti les fichiers dans 2 branches : **src/main/java** et **src/main/resources**.
- Le code binaire qui est généré par la compilation est placé dans la branche **target/classes**.

Tout ceci a été organisé à la main, dans Eclipse, sans utiliser Maven.

Comme vous le voyez dans le volet de gauche d'Eclipse, l'emplacement des bibliothèques est indiqué à Eclipse par des User Libraries.

Le travail qui vous est demandé ici, est de transformer ce projet en projet Maven et de supprimer l'utilisation des User Libraries. Les bibliothèques seront identifiées grâce à un fichier **POM**.

Transformation du projet en projet Maven

- Faites un clic-droit sur le projet **exo-3-maven**, puis sélectionnez "Configure > Convert to Maven Project".
- Dans la boîte de dialogue, indiquez les paramètres suivant :
  - *Group Id :* **`fr.3il.ei`**
  - *Artifact Id :* **labofx**
  - *Version :* **`4.0.0-SNAPSHOT`**

  Puis, actionnez le bouton "Finish".

Cette opération a créé un fichier **pom.xml** qui, pour le moment, ne contient aucune dépendance. Cela va être votre travail de les ajouter.

Ajout des dépendances dans le fichier POM

Suppression des User Libraries

- Dans le volet de gauche d'Eclipse, observez les User Libraries qui ont été affectées au projet **exo-3-maven**. Attention ! Le JRE n'en fait pas partie.

  Il y a donc 3 User Libraries :
  - **Annotations Java EE**
  - **JavaFX**
  - **Spring**

Votre mission consiste à remplacer ces User Libraries par des dépendances dans le fichier **POM** afin d'obtenir un résultat équivalent.

- Dans le volet de gauche, au niveau du projet **exo-3-maven**, supprimez les 3 User Libraires. Pour cela, vous pouvez utiliser le menu contextuel :
  "Build Path > Remove From Build Path".

  Attention ! Ne supprimez pas le **JRE**.

- Immédiatement, des erreurs apparaissent.
  Vous allez les faire disparaître en utilisant Maven, lors des prochaines étapes.

Préparation du fichier POM

- Faites en sorte que le fichier **POM** soit ouvert dans la vue centrale d'Eclipse et choisissez l'onglet "pom.xml" au bas de cette vue.
- Placez-vous tout à la fin du fichier et insérez une ligne vide juste avant la balise **</project>**.
- Faites le nécessaire pour que, sur cette ligne, soit écrit le code :
  ```
  <dependencies></dependencies>
  ```
- Puis, insérez une ligne vide entre ces 2 balises **<dependencies></dependencies>** .
  C'est à cet endroit, entre la balise ouvrante **<dependencies>** et la balise fermante **</dependencies>**, que vous allez insérer les dépendances aux différentes bibliothèques.

Ajout des dépendances dans le fichier POM

Il n'est pas facile d'utiliser Maven dans les conditions où vous êtes, car vous n'avez pas accès à Internet.

On a donc reproduit en local des conditions qui simulent les ressources auxquelles vous auriez accès grâce à Internet.

- Dans le volet de gauche d'Eclipse, dépliez le projet **~MVNRepository**.
  Ce projet contient des pages HTML qui ont été capturées sur le site web **https://mvnrepository.com**
- Parmi ces pages HTML, identifiez en une qui correspond à une bibliothèque dont votre projet a besoin.
- Double-cliquez sur la page pour l'ouvrir dans un navigateur web et récupérez-y les informations dont vous avez besoin.
- Ensuite, revenez à Eclipse et faites le nécessaire pour ajouter la dépendance dans le fichier **POM**.
- Ensuite, recommencez. Ajoutez les bibliothèques qui vous paraissent nécessaires pour que l'application puisse fonctionner.
  Attention ! Une pénalité sera attribuée si vous ajoutez des bibliothèques inutiles.

Test du fonctionnement

- Lorsque vous avez ajouté toutes les dépendances nécessaires dans le fichier **POM** : enregistrez les modifications du fichier **POM**.
  À la fin, il ne doit plus y avoir aucune erreur.
  S'il y en a, c'est qu'il doit manquer une bibliothèque Tâchez de l'identifier et ajoutez-la au fichier **POM**.
- Démarrez l'application en exécutant la classe **AppliLaboFX** pour vérifier que tout fonctionne correctement.

Ajout d'une classe de test

- Observez qu'il y a, à la fin du projet **exo-3-maven**, trois dossiers nommés **src**, **target** et **test**.
- Faites le nécessaire pour déplacer le dossier **test** dans le dossier **src**.
- Une nouvelle branche **src/test/java** a dû apparaître et celle-ci contient une erreur.
  En effet, cette branche contient une classe de test basée sur le framework JUnit.
  Faites disparaître cette erreur en ajoutant au fichier **POM**, la bibliothèque nécessaire.

L'erreur doit disparaître et il doit être possible d'exécuter la classe de test **TestLaboFX**.

4. JUnit

Le projet **exo-4-junit** contient :

- Une classe **Produit** qui est de type "Objet Métier", c'est-à-dire qu'elle ne contient pas de traitements, mais uniquement les données qui décrivent un produit : id, nom, prix et quantité en stock.
- Une classe **ModelStock**. Elle permet de gérer une liste de produits (qui constitue un stock) et d'effectuer différentes opérations sur cette liste.
- Une classe **TestModelStock**. C'est une classe de test, basée sur JUnit, dont l'objectif est d'effectuer des tests unitaires sur la classe **ModelStock** afin de vérifier qu'une future modification de son code n'entraînera pas de régression.

Votre travail consiste à écrire le code de la classe de test.

test1_getNombreProduits

Méthode initialiserStock()

- Dans la branche **src/test/java**, ouvrez le code de la classe **TestModelStock**.
  Une partie de son code a déjà été écrit, mais il est inachevé.
- Observez la méthode **initialiserStock()**.
  Cette méthode reçoit en paramètre une variable **stock** qui représente une liste de produits.
  La méthode supprime le contenu de cette liste et y insère 4 objets **Produuit**.
  Cela permet d'obtenir un stock dont le contenu est parfaitement connu. Ce qui va nous permettre de l'utiliser pour effectuer des tests.

Méthode test1_getNombreProduits()

La classe **ModelStock** comporte une méthode **getNombreProduits()** qui permet de savoir combien de produits différents sont gérés dans le stock.

Par exemple dans le stock qui est créé par la méthode **initialiserStock()**, il y a 4 produits différents.

- Observez le code de la méthode **test1_getNombreProduits()**.
  Elle contient déjà 4 instructions qui permettent de créer :
  - Un objet **stock** qui est une liste de produits dont le contenu a été inséré par la méthode **initialiserStock()**.
  - Un objet **model** que l'on a initialisé en lui ajoutant le contenu de l'objet **stock**.
    Le **model** contient donc une liste de 4 produits.
- Faites le nécessaire pour que la méthode **test1_getNombreProduits()** soit considérée par Junit comme une méthode de test.
- Complétez son code pour qu'il vérifie que, si on exécute la méthode **getNombreProduits()** de l'objet **model**, on obtient bien la valeur 4.

- Exécutez la classe de test (avec JUnit) pour vérifier que le test fonctionne et qu'il réussit.

test2_ajouterProduit() et test3_supprimerProduit_OK()

Dans les méthodes correspondant aux tests n°2 et 3, on a mis une instruction qui :
- Dans un cas, ajoute un produit supplémentaire au modèle.
- Dans l'autre cas, supprime un produit du modèle.

Les tests à effectuer consistent simplement à vérifier que le nombre de produits contenus dans le modèle a changé. Dans un cas, il vaut 5, dans l'autre il vaut 3.

- Apportez les adaptations nécessaires aux méthodes **test2_ajouterProduit()** et **test3_supprimerProduit_OK()** pour qu'elles effectuent les tests demandés.
- Exécutez la classe de test et vérifiez qu'il y a bien 3 tests qui ont été effectués et qu'ils ont tous réussi.

Mutualisation de code

Élimination des redondances de code

Comme vous l'avez remarqué, les méthodes de test commencent toutes par la même série de 4 instructions qui créent un objet **stock** et un objet **model**.

On souhaite éviter cette redondance de code. Voici quelques conseils

- Créez 2 variables privées au niveau de la classe : une pour le **stock** et l'autre pour le **model**.

  Rappel : au niveau de la classe, il peut y avoir des variables statiques et des variables d'instance (non statiques). Ici, le choix n'est pas évident. Il faudra peut-être procéder par tâtonnements.
- JUnit permet de définir des méthodes auxiliaires qui sont exécutées :
  - soit au tout début, avant le tout premier test.
  - soit de façon répétitive, avant chaque test.

  Vous pouvez utiliser l'une de ces techniques (ou les deux à la fois) pour éviter d'avoir à écrire systématiquement les 4 premières instructions de chaque méthode de test.

  Attention à bien utiliser les variables **stock** et **model** définies dans la classe et à ne pas redéfinir des variables locales.
- À la fin, supprimez les 4 premières instructions de chaque méthode de test.
- Exécutez la classe de test et vérifiez que les 3 tests ont bien été effectués et qu'il ont tous réussi.

Optimisation du code mutualisé

Selon la stratégie que vous avez adoptée, plusieurs cas peuvent se présenter. L'objectif ici est de trouver la stratégie optimale.

- Observez la vue "Console" d'Eclipse.
  Celle-ci contient un certain nombre de fois les messages.
  ```
  Stock initialisé !
  new ModelStock
  ```
- Comptez le nombre de chacun de ces messages.
  S'il y en a plus de 3 pour l'un ou l'autre de ces messages, c'est vraiment trop.
  Faites le nécessaire pour qu'il y ait au maximum 3 exemplaires de chaque message.

En fait, le contenu de la variable **stock** n'est jamais modifié. Cela paraît donc inutile de l'initialiser 3 fois. Il devrait suffire de l'initialiser une seule fois, au tout début, avant le tout premier test.

- Si, dans la vue "Console", le message `Stock initialisé !` est affiché plusieurs fois, faites le nécessaire pour qu'il ne soit plus affiché qu'une seule fois.

L'affichage qui traduit la stratégie optimale est donc le suivant :

```
Stock initialisé !
new ModelStock
new ModelStock
new ModelStock
```

test4_supprimer_KO()

Dans la classe **ModelStock**, la méthode **supprimerProduit()** reçoit en paramètre le rang du produit à supprimer. Si on lui passe la valeur 7, elle cherche à supprimer le produit qui occupe le rang 7 dans la liste.

Comme il n'y a que 4 produits enregistrés, il n'y a ~~pas~~ de produit de rang 7. Pour le signaler, la méthode doit retourner une exception de type **IndexOutOfBoundsException**.

- Dans la méthode **test4_supprimer_KO()**, écrivez le code qui vérifie que, lorsqu'on exécute :
  model.supprimer(7)
  cela produit une exception de type **IndexOutOfBoundsException**.
- Lorsqu'on exécute la classe de test, les 4 tests doivent réussir.

test5_getProduit()

- Dans la méthode **test5_getProduit()**, écrivez l'instruction :
  Produit produit = model.getProduit(4);
- Puis, écrivez le code qui effectue les vérifications suivantes :
  - La valeur de la variable **produit** n'est pas **null**.
  - La variable **produit** représente un produit dont le nom est "Produit 4".
- Le test n°5 doit réussir.

test6_getValeurProduit()

La classe **ModelCalcul** comporte une méthode **getValeurProduit()** qui recherche un produit à partir de son identifiant et calcule la valeur du stock pour ce produit (en se basant sur le prix et la quantité en stock).

L'idée est de faire le test pour chacun des 4 produits en stock en utilisant un test paramétré afin d'éviter la redondance de code.

Voici les valeurs attendues :

id	valeur
1	249.75
2	0.00
3	0.00
4	106.56

- Configurez la méthode **test6_getValeurProduit()** afin qu'elle soit considérée par JUnit comme un test paramétré appliqué aux 4 couples de données présentés ci-dessus.
- Écrivez le code qui vérifie que lorsqu'on exécute l'instruction
  model.getValeurProduit(id)
  on obtient bien la valeur attendue.
- Le test n°6 doit produire 4 itérations qui réussissent toutes les 4.