TP : Test Unitaires, Mocks et Test Driven Development

Sommaire

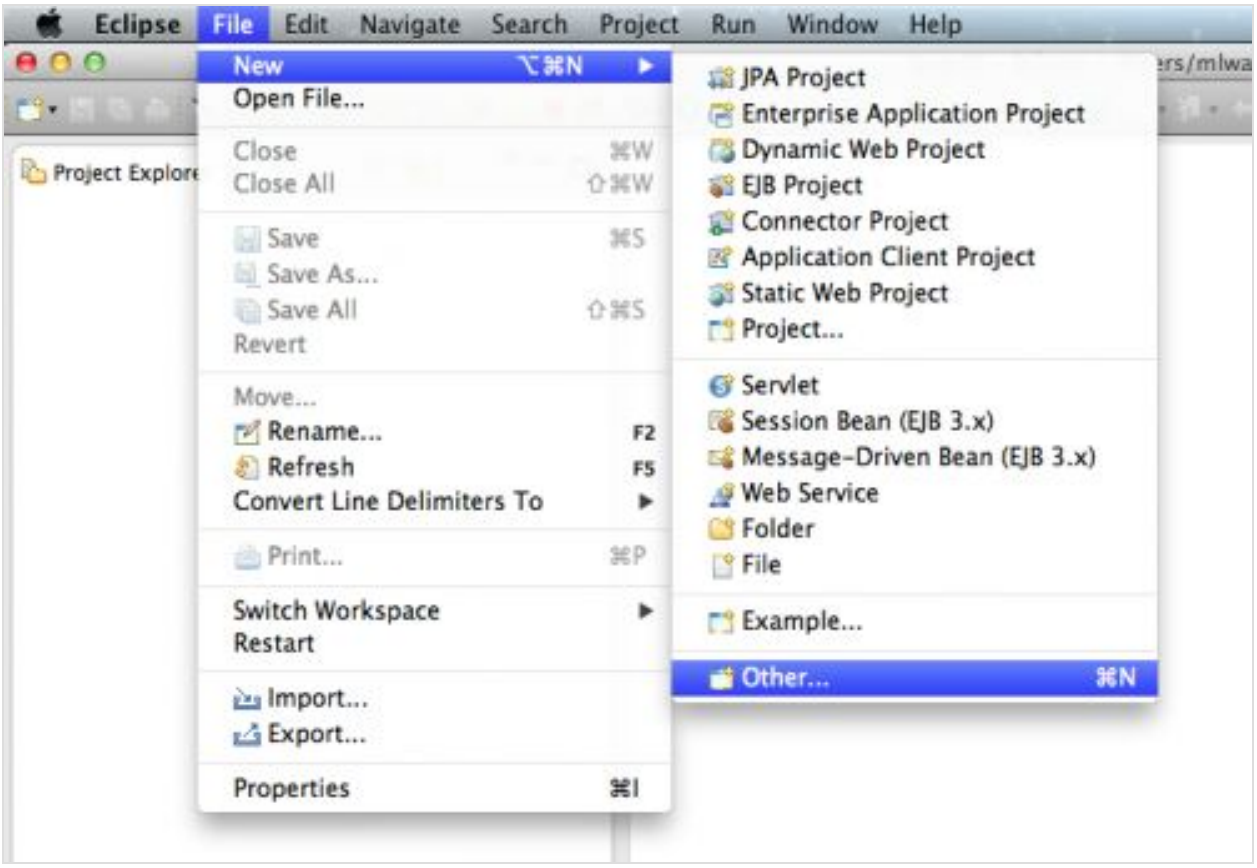
Lors de ce TP nous allons créer des tests unitaires. Vous devrez dans tous les cas respecter la structure verifyWhenThen pour le nommage des méthodes de test unitaire.

1. Création d’un nouveau projet maven dans Eclipse

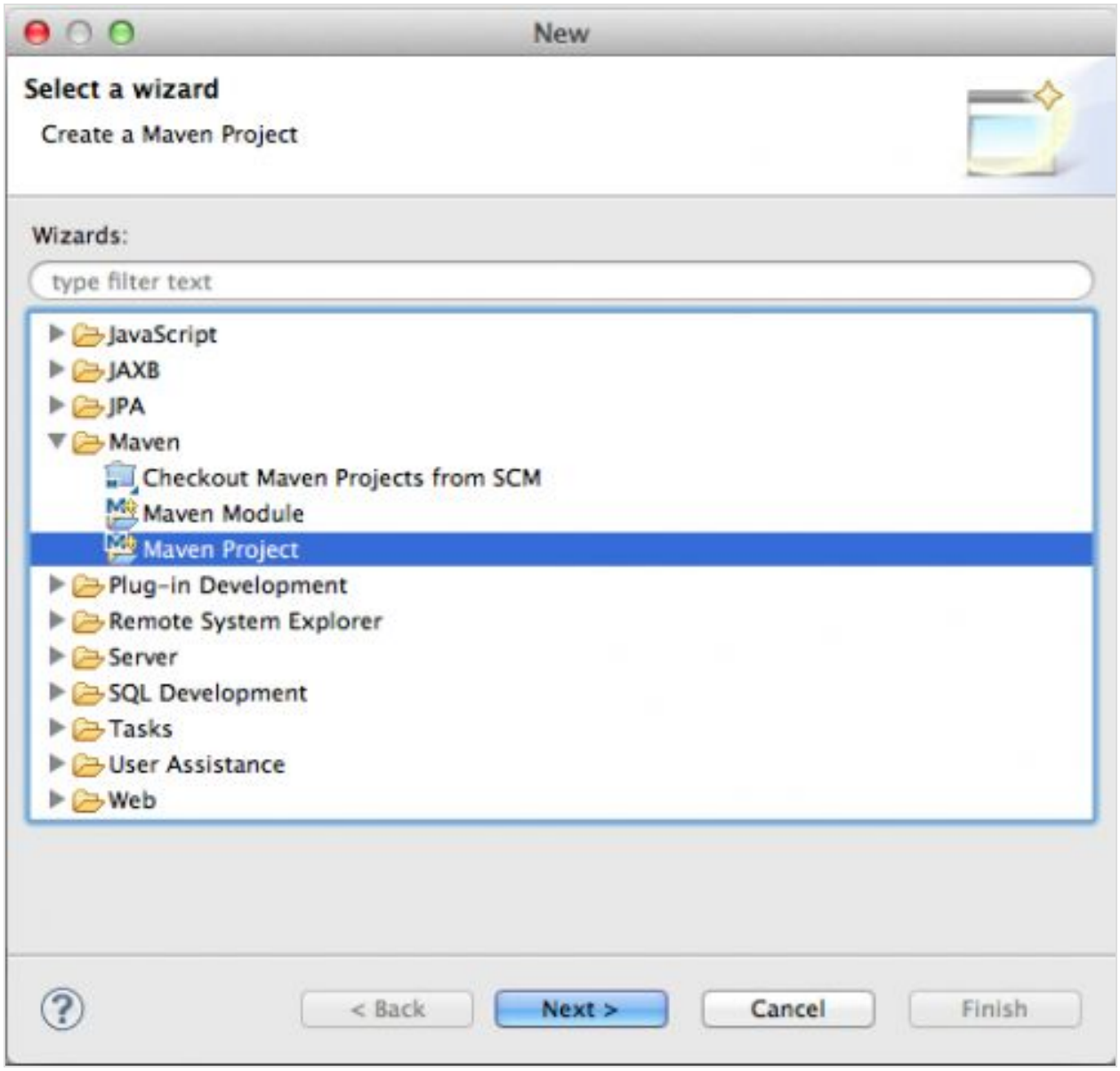
Cette partie servira pour chaque exercice et vous permettera de créer un nouveau projet maven facilement. Lisez-le la première fois et passez à la section suivante.

Lors de la création d'un projet dans Eclipse, on peut utiliser Maven pour gérer plus facilement les dépendances et pour résoudre les dépendances transitives automatiquement.

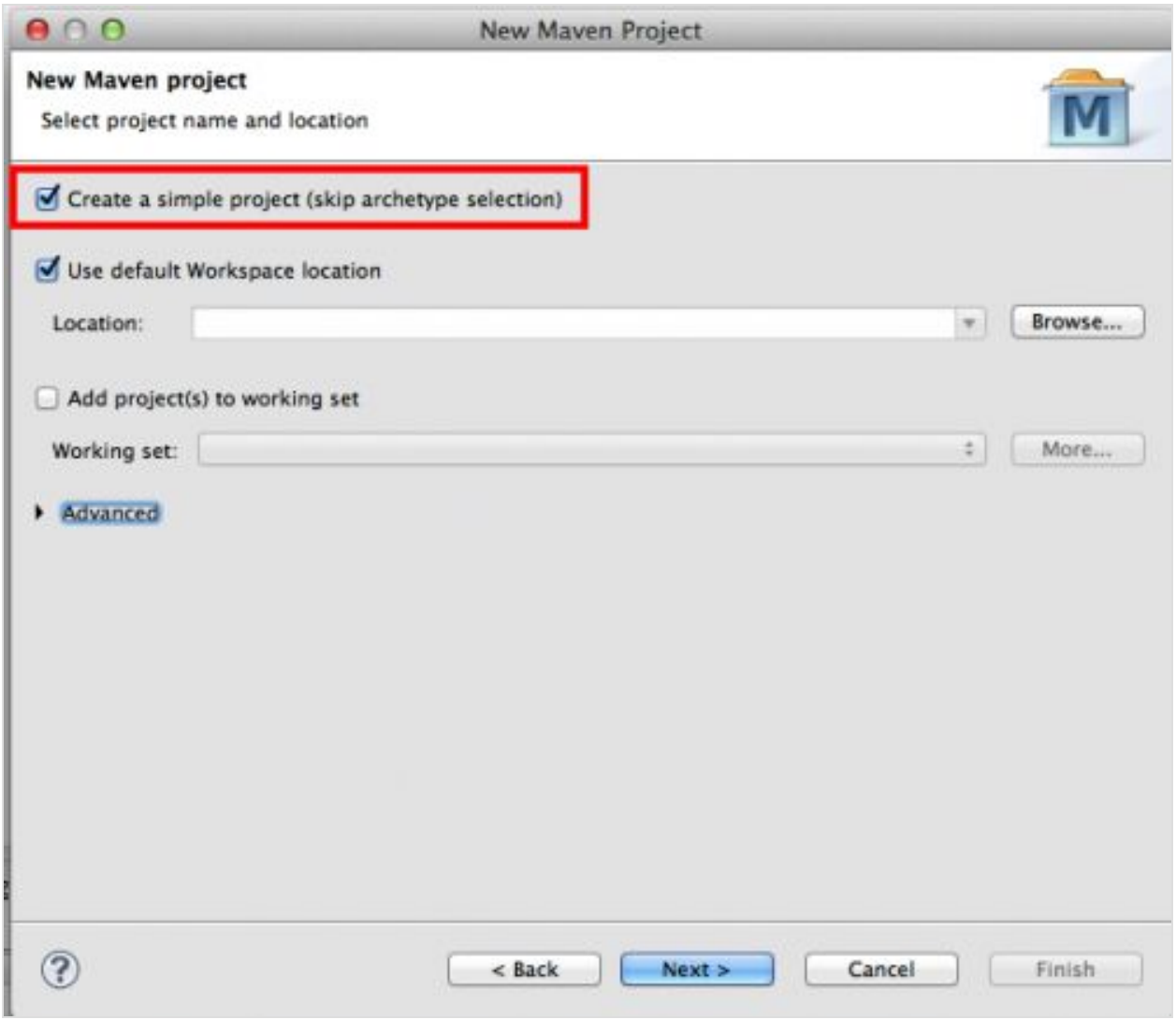
a. Dans **l'IDE Eclipse, accédez à Fichier> Nouveau> Autre ...** dans le but de lancer l'assistant de création de projet.



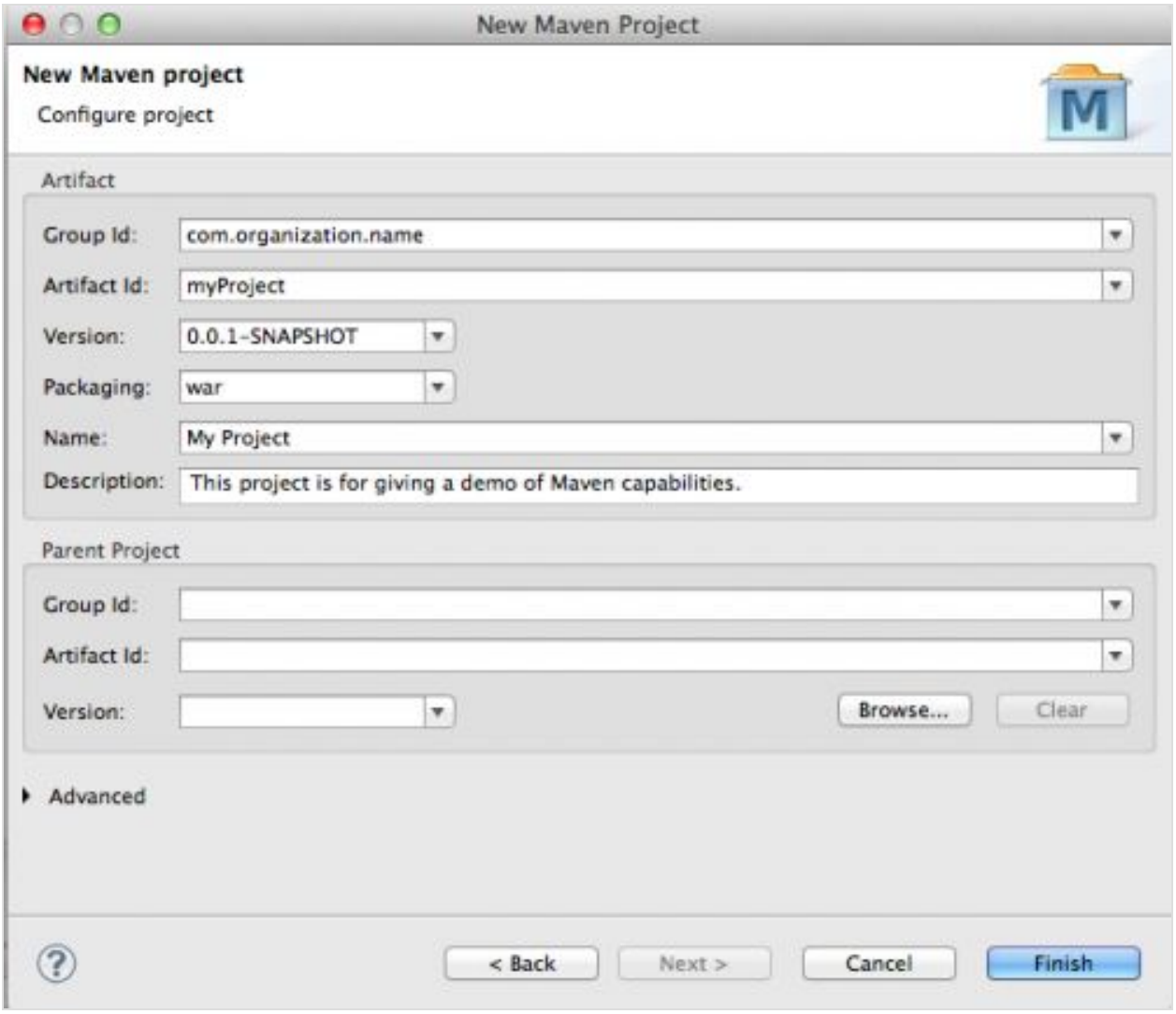
b. Aller jusqu'au dossier **Maven**, ouvrez-le, et choisissez **“Maven Project”**. Ensuite, choisissez Suivant.



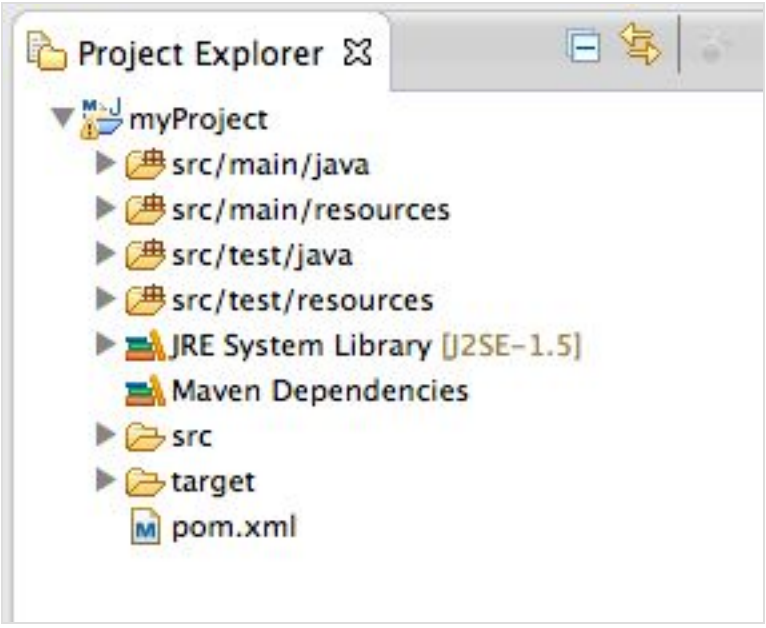
c. Choisissez un type de projet simple “Create a simple project”. Cela va créer un projet Java Maven basique. Si vous avez besoin d'une configuration plus avancée, laissez ce paramètre décoché. Ne pas toucher aux autres options, et cliquez sur Suivant.



d. Maintenant, vous aurez besoin d'entrer des informations concernant le projet Maven que vous voulez créer. Pour plus d'informations, vous pouvez lire la documentation Maven (http://maven.apache.org/pom.html#Maven_Coordinates). En général, le **Groupe Id** doit correspondre au nom de votre organisation, et **l'artefact Id** doit correspondre au nom du projet. La version est à votre discrétion. Si cela est un projet autonome qui ne possède pas de dépendance parente, vous pouvez laisser **la section "Parent Project"** à vide (ce qui est le cas dans notre exercice). Remplissez les informations appropriées, puis cliquez sur Terminer.

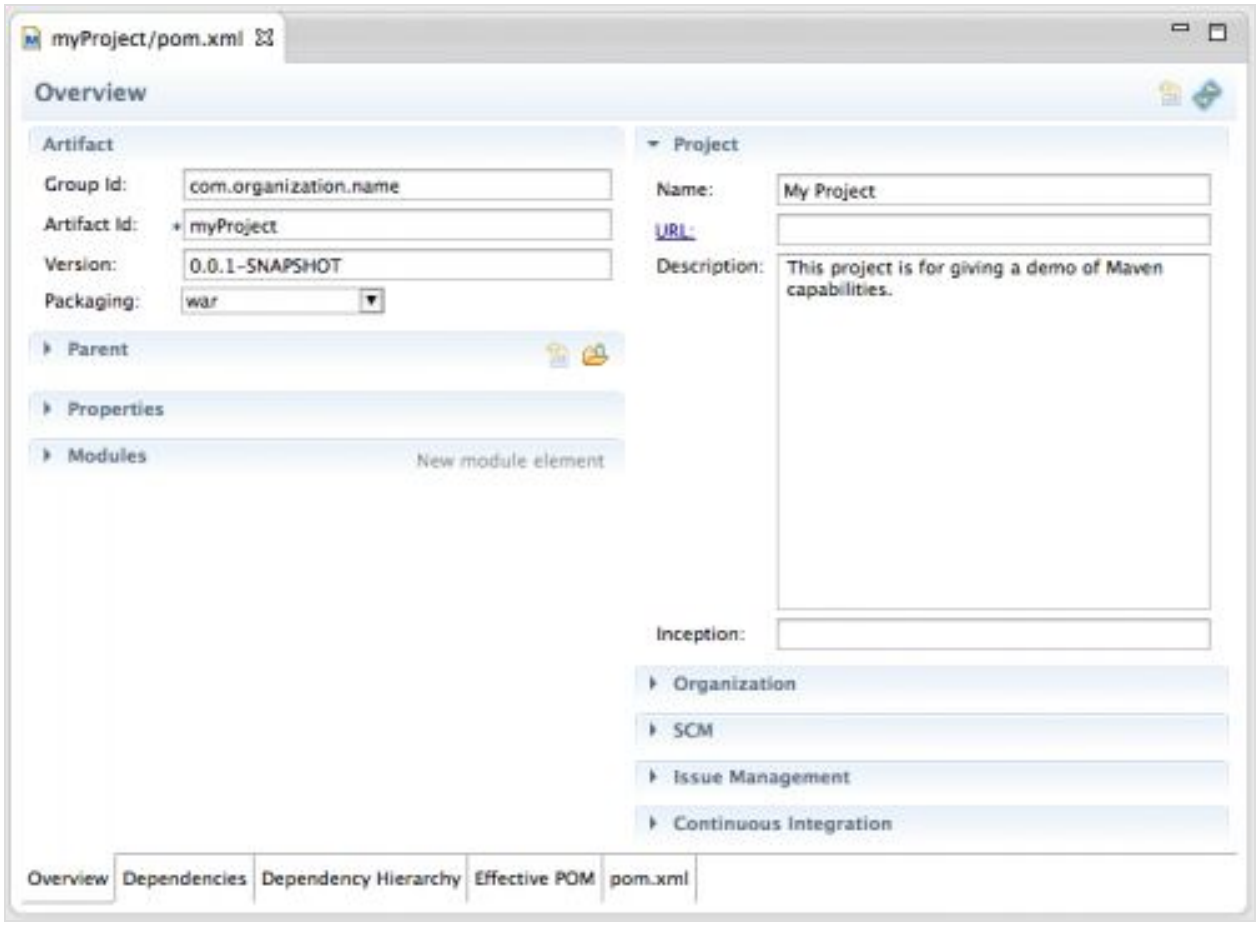


e. Vous remarquerez maintenant que votre projet a été créé. Vous pouvez placer votre code Java **/ src / main /** java, les ressources dans **/ src / main /** resources, et votre code de test et des ressources dans **/ src / test / java** et **/ src / test / ressources** respectivement.



f. Ouvrez le **fichier pom.xml** pour visualiser la structure Maven mis en place. Dans ce fichier, vous pouvez voir les informations saisies à l'étape 4. Vous pouvez également utiliser les onglets au bas de la fenêtre pour changer et pour afficher **les** dépendances, la **hiérarchie des** dépendances, le **POM**, et le code XML brut dans l'onglet **pom.xml**.

Vous avez maintenant un nouveau projet Java avec Maven activé.



g. Maintenant, Cliquez sur l'onglet pom.xml pour visualiser le fichier sous son format original (xml).

h. Ajoutez la dépendance junit en version 4.12 en scope "test" en vous aidant du site https://maven.apache.org/

A inclure dans le fichier pom.xml au bon endroit :

```
<dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
</dependency>
```

i. Vous êtes maintenant prêt pour créer de nouveaux projets maven sous eclipse. Let's play !

2. Test Junit en TLD (Test Last Development)

Dans cette première partie, nous allons implémenter les tests après avoir développer l'implémentation.

Reprenez le chapitre 1 pour créer un projet avec le group id : **fr.enst** et l'artifact id : **customer.** Le projet sera dans la version **1.0-SNAPSHOT.**

Le but de ce test est de vérifier la méthode matchPassword de Customer. Cette méthode vérifie que le mot de passe donné concorde avec celui du Customer.

1. Créer la classe Customer dans le package fr.enst.customer dans le dossier src/main/java
2. Cette classe contiendra les propriétés id et password.
3. Ajouter la méthode matchPassword dans la classe Customer :

```
public boolean matchPassword(final String password1) throws CheckException {
        if (password1 == null || "".equals(password1))
            throw new InvalidPasswordException();

        // The password entered by the customer is not the same stored in database
        return password1.equals(getPassword())
}
```

4. Créer l'exception "InvalidPasswordException" dans le package fr.enst.customer.exception
5. Créer la classe fr.enst.customer.CustomerTest dans le dossier src/test/java du module correspondant.
6. Lancer un build et corriger les erreurs.

Test 1 :

le test vérifie que si le password est null, la méthode renvoie une exception de type InvalidPasswordException

Test 2 :

le test vérifie que si le password est une chaîne de caractère vide, la méthode renvoie une exception de type InvalidPasswordException

Test 3 :

le test vérifie que si le password passé en paramètre n'est pas égale au password du client (Customer), la méthode renvoie la valeur faux.

Test 4 :

le test vérifie que si le password passé en paramètre est égale au password du client (Customer), la méthode la valeur vraie.

3. Utilisation de Mockito pour les mocks en TLD

Dans cette deuxième partie, nous allons concrétiser ce que nous avons vu en cours avec l'utilisation des mocks dans le projet shopping. Nous utiliserons le framework Mockito pour nous faciliter la génération des Mocks.

Reprenez le chapitre 1 pour créer un projet avec le group id : **fr.enst** et l'artifact id : **shopping.** Le projet sera dans la version **1.0-SNAPSHOT.**

Ajoutez la dépendance junit en version 4.12 et mockito en version 1.9.5 dans le scope "test" :

```
<dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-all</artifactId>
        <version>1.9.5</version>
        <scope>test</scope>
</dependency>
```

Créer l'interface CatalogService avec le package "fr.enst.shopping" dans le dossier src/main/java. Cette classe contient une méthode :

```
ItemDTO findItem(Long itemId) throws  CheckException;
```

Créer la classe ShoppingCartServiceImpl avec le package "fr.enst.shopping" dans le dossier src/main/java comme ci-dessous :

```
public class ShoppingCartServiceImpl {

 CatalogService catalogService;

 public CatalogService getCatalogService() {
   return catalogService;
 }

 public void setCatalogService(CatalogService catalogService) {
   this.catalogService = catalogService;
 }

 public FinalCartDTO getCartItems(Map<Long, Integer> map)
     throws CheckException {

   if (map == null){
```

```
            throw new CartIsNullException();
        }


        if (map.keySet().size() == 0){
            throw new CartIsEmptyException();
        }


        final ArrayList<ShoppingCartItemDTO> carts = new ArrayList<ShoppingCartItemDTO>();
        double total = 0.0;

        FinalCartDTO finalCart = new FinalCartDTO();
        finalCart.setCartItems(carts);

        final Iterator<Long> it = map.keySet().iterator();

          while (it.hasNext()) {
             final Long key = it.next();

             final Integer value = map.get(key);

             ShoppingCartItemDTO ciDTO ;
                ItemDTO itemDTO = catalogService.findItem(key);

                // convert catalog item to cart item
                ciDTO = new ShoppingCartItemDTO(itemDTO.getId(),
                        itemDTO.getName(),
                        itemDTO.getProduct().getDescription(),
                        value.intValue(),
                        itemDTO.getUnitCost());

                carts.add(ciDTO);

                total = total + (value.doubleValue() * itemDTO.getUnitCost());

          }
        finalCart.setTotal(total);

          return finalCart;
      }

    }
```

Créer les classes FinalCartDTO, ShoppingCartItemDTO et ItemDTO et les exceptions CheckException, CartIsNullException, CartIsEmptyException pour faire compiler et fonctionner ShoppingCartItemDTO

Faire compiler le tout.

Le but de ce test est de vérifier la méthode getCartItems de la classe ShoppingCartServiceImpl. L'implémentation est fournie. Cette méthode a pour unique argument une Map, structure stockant des objets sous format clé-valeur :

- La clé est l'identifiant de l'item mis en panier
- La valeur correspond au nombre d'items choisis.

La méthode va transformer cette map en un objet complexe appelé FinalCartDTO constitué :

- Du prix global du panier : champ **total**
- D'une liste d'éléments de type ShoppingCartItemDTO : champ **cartItems**.

Premièrement, vous devez créé la classe de Test. A vous de trouver le nom et le bon dossier !

Test 1 :

Lorsque la map passée en argument est null, la méthode renvoie une exception de type CartIsNullException.

Test 2 :

Lorsque la map passé en argument ne contient aucun élément, la méthode renvoie une exception de type CartIsEmptyException.

Test 3 :

Lorsque la map contient un élément, vérifier que la méthode findItem du catalogService est appelée.
Vous devez mocker le service catalogService.

Test 4 :

Lorsque la map contient 1 élément, la méthode renvoie un FinalCartDTO.
Vérifier que ce FinalCartDTO est non null.
Vous devez mocker le service catalogService et simuler sa méthode findItem.

Test 5 :

Lorsque la map contient 3 éléments, la méthode renvoie un FinalCartDTO.

Récupérer les cartItems contenu dans le FinalCartDTO et vérifier que le nombre d’élément de cette liste est égale à 3.
Vous devez mocker le service catalogService et simuler sa méthode findItem.

4. TDD : premiers pas

Dans cette deuxième partie, nous allons développer en étant piloté par les tests.

Reprenez le chapitre 1 pour créer un projet avec le group id : **fr.enst** et l'artifact id : **creditcard.** Le projet sera dans la version **1.0-SNAPSHOT.**

Dans ce projet nous allons développer la méthode verifyNumber de la classe CreditCardVerifyerImpl qui prend en paramètre la chaine de caractère "creditCardNumber".

Cette méthode doit vérifier le numéro de carte bancaire selon l'algorithme de luhn en TDD (http://fr.wikipedia.org/wiki/Formule_de_Luhn#Algorithme)

Vous pouvez vous aider du site : http://www.ee.unb.ca/cgi-bin/tervo/luhn.pl pour alimenter vos tests.

Nous allons écrire les tests suivants les uns après les autres en suivant à la lettre le principe du Test Driven Development :

Attention, lancez vos tests à chaque modification de code (test et implémentation).

Créer une classe de Test dans src/test/java nommée CreditCardVerifyerImplTest en respectant la même arborescence de package que pour la classe d'implémentation

Test 1 :

Ma méthode doit renvoyer une Exception de type NumberFormatException si le numéro de carte envoyé est "AAADE". Ecrire l'implémentation la plus simple qui rend valide le test. Réfactorer.

Test 2 :

Ma méthode doit renvoyer une Exception de type NumberFormatException si le numéro de carte n'est pas composé uniquement de chiffre ou d'espace. Ecrire l'implémentation la plus simple qui rend valide le test. Réfactorer.

Test 3 :

Ma méthode doit renvoyer **OK** si mon numéro de carte est 0. La méthode retourne en effet un objet de type énumération (OK ou KO). Ecrire l'implémentation la plus simple qui rend valide le test. Réfactorer.

Test 4 :

Ma méthode doit renvoyer **OK** si mon numéro de carte est 34. Ecrire l'implémentation la plus simple qui rend valide le test. Réfactorer.

Test 5 :

Ma méthode doit renvoyer **OK** si mon numéro de carte est 042. Ecrire l'implémentation la plus simple qui rend valide le test. Réfactorer.

Test 6 :

Ma méthode doit renvoyer **OK** si mon numéro de carte est 972487086. Ecrire l'implémentation la plus simple qui rend valide le test. Réfactorer.

Test 7 :

Ma méthode doit renvoyer **KO** si mon numéro de carte est 927487086. Ecrire l'implémentation la plus simple qui rend valide le test. Réfactorer.

Test 8 :

Ecrire un autre test avec une numéro qui répond à l'algorithme de Luhn

Test 9 :

Ecrire un autre test avec un numéro qui ne répond pas à l'algorithme de Luhn

Finaliser votre implémentation pour rendre le code plus propre, plus lisible, plus fiable. Vous pouvez modifier l'implémentation à votre guise car les tests sont là pour vérifier que l'implémentation répond correctement.

5. Bonus : TDD à vous de jouer

Dans cette partie c'est à vous de jouer, vous aller devoir implémenter et penser vous même aux tests en utilisant la méthode TDD.

L'objectif est d'implémenter un convertisseur chiffre arabe en chiffre romain. Vous pouvez vous référer à http://fr.wikipedia.org/wiki/Numération_romaine#Exemples_de_chiffres_romains_dans_le_syst.C3.A8me_de_base pour se remémorer la numérotation romaine d'antan.

Reprenez le chapitre 1 pour créer un projet avec le group id : **fr.enst** et l'artifact id : **romanconverter.** Le projet sera dans la version **1.0-SNAPSHOT.**

Le code du converter aura une méthode de ce type :

```
public String convertToRomanNumber(int number)
```

Voici quelques questions à vous poser pour définir vos tests :
La numérotation romaine ne gère pas le 0. Que devez vous faire pour gérer le 0 ?
La notation romaine contient les caractères I, V, X, L, C, D, M. Combien de test devez-vous implémenter pour gérer les cas de bases ?
Comment puis-je gérer et tester des cas complexes tel que 888 ?

Vos tests doivent commencer par tester des cas simples, par la suite des cas de plus en plus compliqués et enfin des cas aux limites.

Faites vous plaisir en *respectant l'esprit du TDD*...