Faculté des Sciences
Aix*Marseille Université

Département Informatique et Interactions
Année 2017-2018

Master 2 FSIL
Fiabilité 1

TP Classes factices

Le but de ce TP est de comprendre comment faire des tests unitaires avec utilisation de classes factices écrites à la main ou générées automatiquement.

1 Simulation manuelle

Dans cette partie nous reprenons ce qui a été vu en TD. Le package *temp* sert à convertir des degrés Celsius en degrés Farenheit et vice-versa. Il comprend 2 classes :

ATester : la classe à vérifier qui contient :

— un attribut privé de type *Conversion*

— une méthode *Double convertit(Double temperature, String sens)* qui prend une temperature, une chaine qui est F2C ou C2F et retourne la conversion de température en Celsius ou Farenheit.

Conversion : la classe de service pour *ATester* qui contient

— une méthode *convF2C* qui convertit une température en Farenheit en Celsius avec la formule : (temp - 32.0) * 5.0 / 9.0,

— une méthode *convC2F* qui convertit une température en Celsius en Farenheit avec la formule : temp * 9.0 / 5.0 + 32.0.

Afin de comparer tous les processus, dans un premier temps :

1. Ecrire les classes `ATester` et `Conversion`.
2. Ecrire la classe de tests `ATesterTest` qui ne teste que la méthode *convertit* de `ATester`.
   Quelques idées : 0°C = 32°F, 100°C = 212°F, 37°C = 98.6°F, -40°C = -40°F.

Ces classes seront dans un projet Java appelé `Normal`. Le)s classes seront dans un package `temperature` (sous répertoire de `src`) et les tests dans un package `testtemperature` (sous-répertoire de `test`).

Simulation via l'héritage

Cré er un nouveau projet appelé `ManualFactice` contenant `src` et `tests`. Créer les packages `temperature`qui contient une classe `ATester` (identique à celle écrite mais pas `Conversion`) et `testtemperature` qui contient une classe `ATesterTest` cope de la précédente.

Réalisation des tests.

1. Lancer les tests : que se passe-t-il ?
2. Renommer `ATesterTest` en `AtesterTestManualMock`. Remplacer `Conversion` par une classe dummy et écrire une classe `MockConversion` simulant `Conversion` en fonction des tests écrits. Effectuer les modifications nécessaires pour que les tests fonctionnent.

Simulation avec inversion de contrôle

On reprend en utilisant l'inversion de contrôle. Supprimer la classe dummy `Conversion` (en la renommant par exemple).

1. Créer une Interface `IConversion` qui permet de donner les fonctionalités attendues de la classe `Conversion`. Ecrire la classes `ATesterWithInterface` et `ATesterWithInterfaceManualMock` similaires aux classes précédentes. Lancer les tests : que se passe-t-il ?
2. Donner une classe factice `MockConversion` qui implémente l'interface `IConversion` et simule le comportement attendu pour les tests. Lancer les tests : que se passe-t-il ?

2 Simulation automatique avec Jmockit

Créer un projet `AutomaticFactice` avec la même organisation que précédemment. Le package `temperature` ne contient que `ATester` (première version sans interface) et éventuellement une classe dummy pour `Conversion` >

2.1 Installation jmockit

Le site `http://jmockit.org/` contient un accès au source et un titorial *jmockit* . Pour installer *jmockit* , récupérer les sources sous format zip puis décompresser le fichier et utiliser `maven` (commande mvn) pour construire le projet (ce qui fournira les jars à inclure dans les propriétés du projet). Si `maven` (version ≥ 3.3 n'est pas installé l'installer. Les groupes n'arrivant pas à créer les jars pourront les acheter auprès de l'enseignant. Un plugin eclipse existe (à installer via le Marketplace d'eclipse) mais il n'est pas nécessaire.

2.2 Présentation

Lire attentivement la documentation pour comprendre comment écrire une classe de tests qui simule une classe non écrite (Conversion dans notre cas) avec *jmockit* . Le répertoire `Devise` contient les classes pour l'exemple du calcul de taux de change entre devises sur lequel vous pouvez vous exercer.

Approche simplifiée L'outil construit les (méthodes des) objets factices à l'exécution et il n'est pas nécessaire de définir les classes factices mais la classe de tests peut utiliser des objets factices de ces classes en les préfixant par `@ Mocked`. Pour le test, il suffit de définir le comportement des méthodes des objets factices. Par exemple, on peut spécifier qu'une méthode est appelée avec certains arguments, renvoie une certaine valeur, est appelée n fois,... Dans les tests on distingue 3 phases :

1. spécification (`Expectations( )...` qui décrit les appels factices attendus (arguments, valeur de retour, nombre d'appel,...) et les enregistre,
2. exécution des méthodes testées puis utilisation des assertions JUnit usuelles pour vérifier que les valeurs attendues sont celles calculées,
3. vérification (`Verifications( ) ...` qui vérifie que l'exécution a bien effectué certains appels (arguments, nombre d'appel,...)

Si les spécifications ou les vérifications ne sont pas satisfaites lors de l'exécution le test echoue de même qu'il échoue si les assertions JUnit ne sont pas satisfaites.

1. Lancer les tests avec la classe de tests fournie après avoir configué votre projet correctement (piège sur l'ordre des jars, lire la doc).
2. Modifier les tests en changeant les spécifications et vérifications puis en les exécutant pour comprendre leur fonctionnement à la lumière des résultats obtenus.
3. Ecrire la classe de tests pour la version avec interface.

Ecriture d'une classe utilisant jmockit Ecrire les classes pour les deux versions de `ATester` (celle sans interface et celle avec interface). Utiliser *jmockit* pour refaire simplement les tests, puis compliquer au fur et à mesure en utilisant le plus de constructions possibles et de combinaisons entre spécifications et verifications. Lire également ce qui est dit dans la documentation sur la couverture de code et le tester.

2.3 Test d'une application plus complexe

On veut écrire et tester une classe `Session` qui gère une interaction avec un distributeur de billet de banque, une carte bancaire et une banque. Un numéro de carte est un nombre à 16 chiffres, un code de sécurité un nombre à 3 chiffres, le code Pin un nombre à 4 chiffres (tous sont positifs).

1. La classe `Distributeur` a les méthodes `int sendPin(numeroSession)` qui renvoie le code Pin d'une carte bancaire, `String envoyerOp(numeroSession)` qui renvoie la chaine correspondant à l'opération bancaire `"debiter", "consulter"`, `void FinSession(numsession)` qui termine la session pour le distributeur.
   La classe `Banque` a les méthodes `Boolean authentifier(long numeroCarte, int codeSecurite)`, `Boolean autoriser(int numeroSession, String operation)`.
   La classe `Carte` a les méthodes `double getNumero()`, `int getPin()`, `int getCodeSecurite()`.
2. La classe `Session` classe a les attributs privés `banque, carte, distributeur, connecté, numsession` passés en paramètres au constructeur (sauf `connecté` initialisé à `false` et les méthodes `startSession()` qui vérifie que `connecte` est `false`, demande l'authentification de la carte à la banque, et si celle-ci est positive met connecté à `true`. La méthode `validerOperation(String op)` demande le code Pin au distributeur, le vérifie s'il est correct puis demande à la banque l'autorisation pour cette opération. Elle renvoie une exception `IllegalOperationException` si elle ne l'est pas. Si le code Pin n'est pas correct, la méthode redemande un code Pin au distributeur (le nombre maximal d'essais pour obtenir le code Pin est 4). Si plus de 4 essais sont effectués, la méthode déclenche une exception `IllegalCardUseException`.

Réaliser la classe `Session` et la tester en utiisant des objets factices *jmockit* pour simuler le comportement de la carte, du distributeur et de la banque. Réduire au minimum la gestion des exceptions (elles n'affichent qu'un message d'erreur).

3 Travail à rendre

Le rendu se fera en deux étapes : un rendu **à la fin du TP** dans l'activité devoir `TPFactice Partie1` et un à effectuer **avant le lundi 24 octobre 23h55** dans l'activité devoir `TPFactice Partie2`. Comme pour le TP précédent, le rendu est une archive zip qui s'appelera `TPFACTICE-PARTIE(1 ou 2)-TPi.zip` et qui crée un répertoire du même nom au désarchivage. **Les fichiers pdf devront respecter le modèle de documents donné sur la page du cours à la rubrique bibliographique**.

1. Le premier rendu contiendra toutes les classes (sources et tests) pour les différentes variantes de `ATester` (avec/sans interface) et les tests sans utiliser *jmockit* .
2. Le deuxième compte-rendu donnera l'application es classes de tests utilisant *jmockit* (pour `Session`) et un petit rapport expliquant l'utilisation faite de *jmockit* et les fonctionalités utilisées ainsi que celles non utilisées mais qui vous paraissent utiles et facilement utilisables.