Faculté des Sciences
Aix*Marseille Université

Département Informatique et Interactions
Année 2016-2017

Master 2 FSIL

Fiabilité Logicielle 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 Travail à rendre

À la fin du TP (11/10 à 17h), vous enverrez par mail un zip contenant vos sources (.java). Vous pouvez ajouter un rapport en pdf répondant aux questions du tp, expliquant votre démarche et/ou des points techniques. Ce Tp est largement réalisable dans le temps imparti.

2 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 en particulier

1. un attribut privé de type *Conversion*
2. 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 (formule: temp-32.0)*5.0/9.0),
- une méthode *convC2F* qui convertit une température en Celsius en Farenheit (formule: temp*9.0/5.0+32.0).

Ecrire la classe *ATester*. **Ne pas** écrire la classe *Conversion* qui est une application développée par d'autres équipes.

Simulation via l'héritage

Réalisation des tests.

1. Ecrire la classe de test *atesterTest* pour *ATester* (quelques idées: 0°C=32°F, 100°C=212°F, 37°C=98.6°F, -40°C=-40°F). Lancer les tests: que se passe-t-il?

2. Expliquer comment remplacer *Conversion* par une autre classe*MockConversion* qui permet d'exécuter les tests. Ecrire la classe *MockConversion* et relancer les tests.

Simulation avec inversion de contrôle

On reprend en utilisant l'inversion de contrôle.

1. Créer une Interface *IConversion* qui permet de donner les fonctionalités attendues de la classe *Conversion*. Ecrire dans les classes *ATesterBis*, *ConversionBis*, *ATesterBisTest* similaires aux classes précédentes. Lancer les tests: que se passe-t-il?

2. Donner une classe factice *MockConversion* qui implémente l'interface *IConversion* et de simuler le comportement attendu pour les tests. Lancer les tests: que se passe-t-il?

3 Simulation automatique avec Jmockit

Récupérer Jmockit sur le site officiel `http://jmockit.org/`. Lire la documentation et installer l'application dans un répertoire personnel. Ajouter les *.jar* dans le projet et/ou dépendence *maven*. L'outil *jmockit* est riche et vous devez lire la documentation pour découvrir toutes ses fonctionalités (les tutoriels GettingStarted et Mocking sont un bon début). Attention, placer JMockit avant JUnit dans le classpath ou rajouter l'annotation `@RunWith(JMockit.class)` avant la classe junit (voir tutoriel *GettingStarted*).

3.1 Présentation

L'outil construit les (méthodes des) objets factices à l'exécution et il n'est pas nécessaire de définir les classes factices. 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) le rejeu qui doit se conformer à la spécification et exécute le code instrumenté (qui appelera les méthodes factices), (3) la vérification (*Verifications( )*) qui vérifie que l'exécution

a bien effectué certains appels factices. On peut également mettre des assertions JUnit dans le code instrumenté. La partie (1) ou la partie (3) peuvent être absentes. *Expectations* et *Verifications* peuvent être strictes ou non et ont des rôles complémentaires. Un mocking stict (voir StrictExpectations et VerificationsInOrder/FullVerifications) attend les appels dans le même ordre que celui fourni.

Un exemple du cours: *calc* est un objet qui implémente une interface avec une méthode *getTaux* qui est déclarée factice dans la classe de test par:

`@Mocked ITauxChange conv;`.

```
@Test
public void testEuro2euroOnce() {
        new Expectations (){{
                conv.getTaux(anyString,anyString);
                result=1.0;
                times=1;
                }
        };
        calc =new DeviseCalcWithInterface(conv);
        double expected=1.0;
        double value=calc.euro2euro(1.0);
        Assert.assertTrue(expected==value);
}
```

3.2 Retour sur la première partie

Reprendre l'exercice sur la conversion en remplaçant les classes factices écrites à la main par l'utilisation de fonctionalités *jmockit* permettant de réaliser les tests. Essayer les différentes possibilités offertes par jmockit, par exemple :

- utilisent soit *Expectations( )*, soit *Verifications( )*, soit les deux.
- mettent en oeuvre une spécification ou une vérification de fonction avec des arguments précis, quelconques (anyInt, anyDouble, anyString), renvoyant un résultat, appelée $n = 0, 1, 2, \ldots$ fois

3.3 Test d'une application carte bancaire

Un terminal de paiement par carte bancaire permet de lire un numéro de carte, de l'authentifier via un service externe de validation, puis au possesseur de la carte de se connecter en entrant son code secret. Si celui-ci est correct, la carte est connectée. Si le numéro de carte

n'est pas légal, le terminal refuse la connection, et si l'utilisateur entre plus de 4 fois un code erroné le terminal invalide la carte.

Il s'agit de tester la classe *connectToTerminal* qui réalise les opérations du terminal. Elle utilisera les services des classes *Card* et *Validator*. Elle contient un attribut privé *connectedCard* de type *Card* (et d'autres à définir) et les méthodes publiques *public boolean validateCardNumber(int cardnumber)* qui interroge son validateur (type *Validator*) et peut déclencher une exception *IllegalCard* et sinon met à jour l'attribut privé *connectedCard*, et la méthode *public void authenticateCode(int secretCode)* qui peut renvoyer une exception *NumberOfTryExceeded* ou *IllegalCard*.

La classe *Validator* contient la méthode publique *Card validateCard(int number)* qui renvoie la carte correspondant au numéro de carte, sinon *null*. La classe *Card* aura les méthodes *boolean isConnected()* renvoie $true$ si la carte est connectée, $false$ sinon, *void setConnection(boolean v)* (établit le statut de connection de la carte à v, *boolean isValid()* qui renvoie la valeur d'un attribut indiquant si la carte est valide ou non, *public void setValidation(boolean v)* qui assigne l'attribue précédent à v, *boolean checkSecretCode(int code)* qui renvoie $true$ si le code secret est égal à *code*, $false$ sinon.

1. Définir les tests nécessaires pour la classe *connectToTerminal*.
2. Écrire la classe *connectToTerminal* et la tester en utilisant jmockit pour simuler les classes *Card* et *Validator*.