Fiabilité Logicielle

Tests : mocking

Arnaud Labourel

2025-26

amU Faculté des sciences Aix Marseille Université

Section 1

Tests avec objets factices

Tests unitaires : problématique

Test unitaire : tester toutes les fonctionnalités d'une classe (en isolation)

Problème : une classe interagit avec d'autres entités

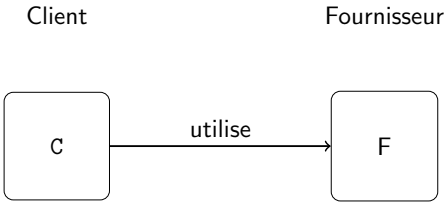
1. Base de données (données persistantes),
2. Service Web, outils réseaux,
3. Classes avec un comportement non-déterministe,
4. Classes de l'application en cours de développement.

L'échec d'un test sur la classe à tester ne doit incomber que de cette classe et non des classes avec qui elle interagit.

Solution

Simuler le comportement des objets hors contrôle.

Schéma classique client/fournisseur



La classe est cliente d'un ou plusieurs fournisseurs.

- `F` pas encore écrite
- Pas de contrôle sur les résultats retournés par `F`

Comment remplacer `F` ?

Introduction : un exemple

Classe `EuroCalc` : convertisseur de valeur en euros dans une autre devise.

- méthodes `euroToDollar`, `euroToEuro`, `euroToYen`, ... qui utilisent la méthode `getRate` d'une classe `CurrencyRate`
- un constructeur `euroCalc(CurrencyRate actualCurrencyRate)`
- un attribut privé de type `CurrencyRate`

Les objets de la classe `CurrencyRate` consulte un site en ligne (accès Web) donnant les cours en temps réel.

Ecrire les **tests** pour `EuroCalc`

Introduction : un exemple (II)

La classe EuroCalc :

```
public class EuroCalc {
  private CurrencyRate rate;
    public EuroCalc(CurrencyRate currentRate){
      this.rate = currentRate;
  }
  public double euroToEuro(double amount){
    // ...
  }
  //...
}
```

Introduction : un exemple de tests

Tester `euroToEuro(double)` taux de change est constant 1.0

```
@BeforeAll
public void setUp() throws Exception {
    rate = new CurrencyRate();
    calc = new EuroCalc(rate);
}
@Test
/* vérifier que euroToEuro retourne la même valeur*/
public void euroToEuroTest() {
    double expected = 1.0;
    double value = calc.euroToEuro(1.0);
    assertThat(expected).isEqualTo(value);
}
```

Introduction : un exemple de tests (II)

Tester `euroToDollar`

- taux fixé à 1.172
- le test vérifie que la valeur est bonne

```
@Test
/* vérifier que la valeur retournée par euroToDollar
   avec un taux de change de 1.172 est correcte */
public void euroToDollar() {
    double expected = 1.172;
    double value = calc.euro2dol(1.0);
    assertThat(expected).isEqualTo(value);
}
```

Exemple : quel est le problème ?

Sur le site web la valeur du taux actuel est utilisé.

- la valeur n'est pas forcément 1.172
- la valeur varie selon le moment du test.

Simuler la classe `CurrencyRate`.

Quand simuler ?

- comportement non-contrôlable (non prédictible, non-reproductible) : évolutif dans le temps, configuration, réseau local, application web, ...
- application incomplète : modules non finalisés, méthodes incomplètes (parfois la classe sous test elle-même !),
- applications coûteuses, états du système difficiles à reproduire, effets de bords, données persistantes.

Comment simuler ?

- Méthodes **manuelles** : écriture de classes permettant de réaliser les tests.
- Méthodes **automatisées** : génération **automatique** de classes permettant de réaliser les tests.

Outils : librairie `mock` en python, `mockito` pour Java, ...

Que simuler ?

- Comportements calculatoires :
  - bons paramètres,
  - bons résultats.
- Appels des méthodes :
  - bonne méthode,
  - nombre d'appels.
- Déclenchement d'exceptions

Introduction : terminologie

Terminologie : **Dummy**, **Stubs**, **Fake**, **Spy**, **Double**, **Mock** (Bidon, doublure, bouchons, factice, ...)

Idée simple : simuler une classe utilisée de manière plus ou moins précise.

Technologie compliquée pour les outils.

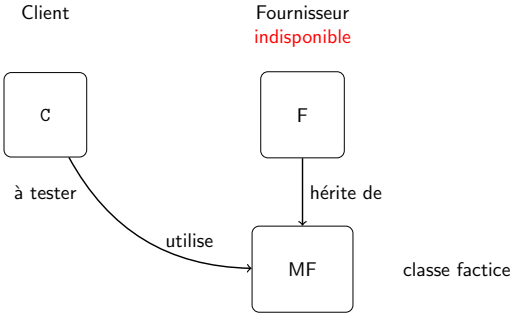
Terminologie

1. **dummy** : objet sans fonctionnalité
2. **stub** (bouchon) : renvoie une valeur fixée par méthode
3. **fake** (simulateur) : implémentation partielle qui renvoie toujours les mêmes valeurs selon les paramètres.
4. **spy** (espion) : vérifie l'utilisation des méthodes.
5. **mock** (doublure) : spy + fake.

Simulation manuelle sur l'exemple

Classe `MockCurrencyRate` remplace/simule `CurrencyRate`

- Même type $\Rightarrow$ hérite de `CurrencyRate`
- Calcule avec le taux voulu $\Rightarrow$ redéfinition des opérations/méthodes.



Simulation manuelle sur l'exemple (II)

Premier essai :

Classe de test :

`rate = new MockCurrencyRate();` au lieu de `rate = new CurrencyRate();`

Classe factice :

```
public class MockCurrencyRate extends CurrencyRate{
    public double getRate(String from, String to){
        return 1.172;
    }
}
```

Satisfaisant (pour les tests envisagés) ?

Simulation manuelle sur l'exemple (III)

Deuxième essai :

```
public double getRate(String from, String to){
     if (from.equals(to)){
             return 1.0;}
     else if (from.equals("euro") && to.equals("dollar")){
                return 1.172;}
             else {return 0.0;}
}
```

Simulation manuelle sur l'exemple : premier bilan

- Aucun changement dans la classe à tester.
- Les tests utilisent des objets de type `MockCurreencyRate` et pas `CurrencyRate`.
- La classe `MockCurrencyRate` est définie en fonction des tests qu'on souhaite faire

Ecrire deux classes (`Test` et `Mock`) au lieu d'une.

Simulation manuelle sur l'exemple : nouvelle problématique

Important : Afin de simuler un objet de type `F`, il est crucial que si `C` utilise un objet de type `F`

- la classe `C` ne doit pas le créer,
- `C` et `F` crées indépendamment et l'objet `F` est fourni (injecté) à `C`

Mais

- Créer une classe quasi-vide `F` pour que `MockF` en hérite
- La classe de test `CTest` dépend d'une implémentation de `MockF` (fixée)

Application des principes SOLID

Solution : inversion des dépendances / du contrôle

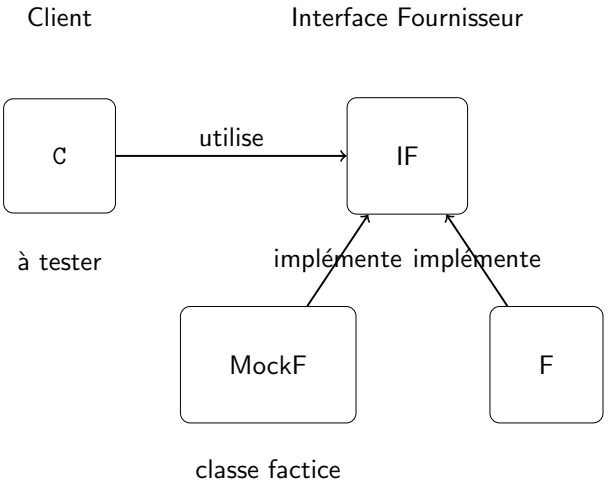
Simulation manuelle sur l'exemple : évolution

- définition d'une interface `IF` que `F` ainsi que `MockF` utiliseront.
- la classe de test `CTest` utilisera l'interface `IF`
- la classe `C` utilisera l'interface `IF`

Ainsi,

- vis-à-vis de `C`, `F` et `MockF` sont interchangeables
- possibilité d'injection de la classe `MockF` dans `CTest`

Simulation manuelle sur l'exemple : évolution (II)



Simulation manuelle sur l'exemple : évolution (III)

L'interface :

```
public interface ICurrencyRate{
  double getRate(String from, String to);
}
```

La classe :

```
public class EuroCalc {
  private ICurrencyRate rate;
  public EuroCalc(ICurrencyRate r){
    this.rate = r;
  }
  ...  // inchangé
}
```

Simulation manuelle sur l'exemple : évolution (IV)

La classe de test :

```
public class EuroCalcTest{
  private ICurrencyRate r;
  private EuroCalc calc;

  @BeforeAll
  public void setUp() throws Exception {
    r=new MockICurrencyRate();   // injection de code possible
    calc=new EuroCalc(r);
  }
}
```

Automatisation : quelques outils

1. EasyMock http://easymock.org
2. JMockit http://jmockit.github.io
3. Mockito https://site.mockito.org

Automatisation : Mockito

Principes de Mockito :

1. créer les objets mocks.
2. décrire leur comportement.
3. à l'exécution, les interactions sont enregistrées
4. à la fin des tests on interroge les objets mock pour savoir comment ils ont été utilisés.

Utilisation : ajout de jar dans les propriétés du projet ou via gradle

Automatisation : écriture de tests

Ajouter `mockito` au projet (jar, build gradle).

Dans la classe de test.

1. Définir les objets simulés :

```
o = mock(Toto.class)
```

2. Spécifier le comportement :

```
when(toto.foo(arg1,...,argn)).thenReturn(resultat)
```

3. Spécifier le déclenchement d'exceptions :

```
doThrow(new MonException()).when(mock).foo()
```

Automatisation : écriture de tests (II)

1. Vérification : nombre d'appels `times()`, pas d'appel `never()`, au moins un `atLeastOne()`, ...

```
verify(mock, times(n)).foo(arg1,...,argn)
```

2. Valeurs anonymes : `anyDouble()`, `anyString()`, ... pour désigner une valeur quelconque du type.
3. Spécification de plusieurs appels :

```
when(appel).thenReturn(resultat1, resultat2)
```

Automatisation : écriture de tests (III)

```java
@BeforeEach
void setUp() throws Exception {
    currencyRate = mock(CurrencyRate.class);
}

@Test
public void euroToDollarRateTest() {
    when(currencyRate.getRate(1.0,"euro","dollar"))
      .thenReturn(1.172);
    calc = new EuroCalc(currencyRate);
    double value = calc.euroToDollar(1.0);
    double expected = 1.172;
    assertThat(value).isEqualTo(expected);
}
```

Automatisation : écriture de tests (III)

```
@Test
public void euroToDollarVerifyCallTest() {
    when(currencyRate.getRate(1.0,"euro","dollar"))
        .thenReturn(1.172);
    calc = new EuroCalc(currencyRate)
    double value = calc.euroToDollar(1.0);
    verify(currencyRate, times(1)).getRate(anyDouble(),
          anyString(), anyString());
}
```

Section 2

Fuzzing

Tests fonctionnels de robustesse

Idée

Tester de manière intensive la robustesse d'un produit fini, une librairie ou des protocoles réseaux dans le but de le faire planter.

Fuzz testing (fuzzing)

- de manière intensive = génération automatique des données de test.
- initialement basé sur des tests aléatoires.

Vise surtout les "corner cases"

Test de *fuzzing* avant tout *release* d'application chez Microsoft, Google, ...

Utilisé pour la première fois en 1988 pour tester des commandes Unix.

Fuzzing : production des données de tests

- statique : par exemple, on parcourt incrémentalement un espace de valeur
- aléatoire : simple à mettre en place, mais couteuse si les cas d'erreur sont en petit nombre
- mutation :
  - on crée un jeu de tests initial
  - on opère des modifications sur les données (mutation) selon des transformations prédéfinies (appliquées possiblement avec une dose d'aléatoire - algorithmes génétiques).

Guide de la génération

- génération de valeurs guidée par une grammaire (définissant les entrées valides) :
  - REGEXP : expressions régulières
  - DTD : *Document Type Definition*
  - XML Schemas : langage de description de format de document XML
- génération guidée par l'exécution symbolique (CFG : *context-free grammar*) : *white-box fuzzing*

Quelques outils de fuzzing

- Radmasa (outil de mutation des données) : https://gitlab.com/akihe/radamsa
- Atheris par Google (outil de mutation des données en Python) : https://github.com/google/atheris
- American Fuzzy Lop par Google (fuzzing boite blanche avec utilisation de métriques de couverture) : https://github.com/google/AFL
- Jazzer (*fuzzing* pour Java) : https://github.com/CodeIntelligenceTesting/jazzer