

Fiabilité Logicielle Test d'Intégration

- 1 Principes
- 2 Objets factices
 - Réalisation d'une classe factice
 - Ecriture de Classe Factice
- 3 Principes
 - Inversion de Contrôle
 - Fabrique d'objet
- 4 Automatiser les Tests avec des Objets Factices

- Test **unitaire**: test de chaque méthode de la classe.
- Test **d'intégration**: tester que des composants (packages par exemple) interagissent correctement.

⇒

Simuler le comportement des autres parties sans attendre leur finalisation.

- à la main.
- de manière automatique.

- Test **unitaire**: test de chaque méthode de la classe.
- Test **d'intégration**: tester que des composants (packages par exemple) interagissent correctement.

⇒

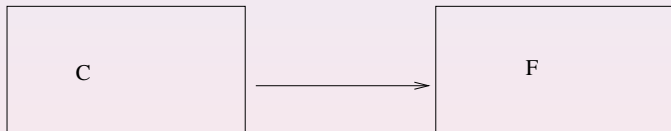
Simuler le comportement des autres parties sans attendre leur finalisation.

- à la main.
- de manière automatique.

Schéma classique client/fournisseur

client

fournisseur



a tester

renvoie des valeurs

Remplacer F?

- F pas écrite
- Contrôle sur les résultats retournés par F

Objets Factices

Un exemple

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

- méthodes *euro2dol*, *euro2euro*, *euro2yen*,...
- qui utilisent la méthode *getTaux* d'une classe *TauxChange*
- un attribut privé de type *TauxChange*
- constructeur *EuroCalc(TauxChange tauxchange)*

TauxChange consulte par exemple un site en ligne (accès Web) donnant les cours en temps réel ou un fichier mis à jour régulièrement.

Ecrire les **tests** pour *EuroCalc*

Tester *euro2euro()*

Le taux de change est constant *1.0*

```
@Test
/*
 * vérifier que euro2euro retourne la meme valeur
 */
public void euro2euro() {
    double expected=1.0;
    double value=calc.euro2dol(1.0);
    Assert.assertTrue(expected==value);
}
```

Tester *euro2Dol*

- Taux fixé à 1.512
- Le test vérifie que la valeur est bonne

```
@Ignore
```

```
/*
```

```
 * vérifier que la valeur retournée par euro2dol
```

```
 * avec un taux de change de 1.512 est correcte
```

```
*/
```

```
public void rate1512euro2dol() {  
    double expected=1.512;  
    double value=calc.euro2dol(1.0);  
    Assert.assertTrue(expected==value);  
}
```

Quel est le problème?

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

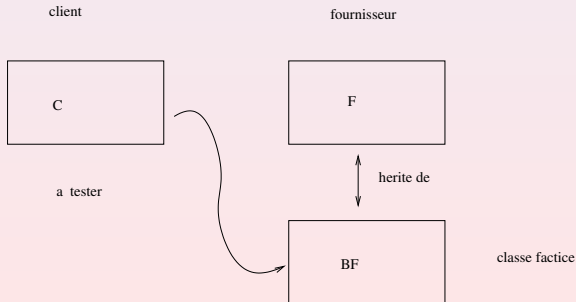
- La valeur n'est pas forcément 1.512
- La valeur varie selon le moment du test.

Solution: simuler la classe *TauxChange*.

Solution: classe factice (bidon)

Classe *MockTauxChange* remplace *TauxChange*
Demandé:

- Même type \Rightarrow Hérite de *TauxChange*
- Calcule avec le taux voulu \Rightarrow redéfinir les opérations.



Premier essai:

```
public class MockTauxChange extends TauxChange{  
  
    public double getTaux(String from, String to){  
        return 1.512;  
    }  
  
}
```

Deuxième essai:

```
public double getTaux(String from, String to){  
    if (from.equals(to)){  
        return 1.0;}  
    else if (from.equals("euro")&&to.equals("dollard"))  
        return 1.512;  
    }  
    else {  
        return 0.0;  
    }  
}
```

from.equals("euro") ou *from=="euro"*?

Ce qui a changé:

- Aucun changement dans la classe à tester.
- Les tests utilisent des objets de type *MockTauxChange* et pas *TauxChange*.
- La classe *MockTauxChange* est définie en fonction des tests qu'on veut faire
⇒ Ecrire deux classes (Test et Mock) au lieu d'une.

Question

Comment modifier de manière à ce que le taux de change soit un paramètre.

Séparer création d'objet et écriture de classe.

Question

Comment modifier de manière à ce que le taux de change soit un paramètre.

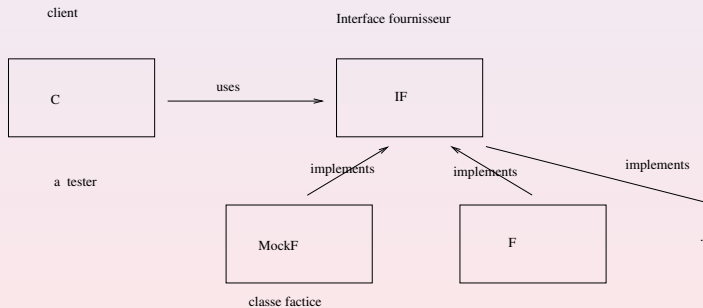
Séparer création d'objet et écriture de classe.

Inversion de controle.

Principe: C utilise un objet de type F

- C ne doit pas le créer
- C et F créés independemment
- communication via **une interface IF** (que F implémente)
- Test avec F ou une classe bidon BF

Le nouveau Schéma



Ce qui change:

- L'écriture de la classe à tester: type **IF** au lieu de **F**.
- L'écriture de la classe factice **MockF** : elle implémente **IF**
- L'écriture de la classe **F**: elle implémente **IF**
- L'écriture de la classe de test **CTest**: type **IF** au lieu de **F**.

Pire qu'avant?

L'**interface**:

```
public interface ITauxChange {  
    double getTaux(String from, String to);  
}
```

La **classe**:

```
public class EuroCalc {  
  
    private ITauxChange conv;  
  
    public EuroCalc(ITauxChange c){  
        this.conv=c;  
    }  
    ... inchangé  
}
```

La classe de **test**:

```
public class EuroCalcTest{  
  
    private ITauxChange c;  
    private EuroCalc calc;  
  
    @Before  
    public void setUp() throws Exception {  
        c=new MockITauxChange();  
        calc=new EuroCalc(c);  
    }  
}
```

Isoler la création d'objet

Principe: un objet ne doit pas **créer** les objets qu'il **utilise**.

Si **C** utilise **F** alors **C** et **F** doivent être créés indépendamment et **F** passé en paramètre à **C**.

Isoler la création d'objet

Principe: un objet ne doit pas **créer** les objets qu'il **utilise**.

Si **C** utilise **F** alors **C** et **F** doivent être créés indépendamment et **F** passé en paramètre à **C**.

- La création d'objet est déléguée à une classe *Fabrique*.
 - Créer un objet=appel à la classe *Fabrique*
 - Ou bien créer un attribut de type *Fabrique*.

Encore mieux: utiliser une interface *IFabrique*

- Permet plus de généricité.
- Pour tester: une fabrique factice pourra fabriquer des objets factices.

Automatiser la création d'objets factices

Utilisation d'outils:

- JMock
- EasyMock
- ...

pour écrire des tests sans écrire les classes factices.

en **spécifiant** le **comportement** des objets factices dans le test.

Automatiser la création d'objets factices

Utilisation d'outils:

- JMock
- EasyMock
- ...

pour écrire des tests sans écrire les classes factices.

en **spécifiant** le **comportement** des objets factices dans le test.

JMock

Principes de base pour écrire les tests:

- Créer une fabrique d'objets factices.
- Créer des objets factices via la fabrique.
- Spécifier le comportement attendu des objets.
- Ecrire les assertions.

Exemple

Création d'une **fabrique d'objets** *mockfactory*

```
@Before
```

```
public void setUp() throws Exception {  
    mockfactory= new JUnit4Mockery();  
}
```

Création de l'**objet factice** *tauxchange*

```
@Test
```

```
public void testEuro2euro() {  
    final ITauxChange tauxchange  
    = mockfactory.mock(ITauxChange.class);
```

Spécification du **comportement** (dans le test):

- La méthode *getTaux* de *tauxchange* est appelée au moins une fois avec les arguments "euro" et "euro"
- Le résultat renvoyé est 1.0

```
mockfactory.checking(new Expectations() {{  
    atLeast(1).of(tauxchange).getTaux("euro","euro");  
    will(returnValue(1.0));  
}});
```

Le test:

```
calc= new EuroCalc(tauxchange);  
double expected=1.0;  
double value=calc.euro2euro(1.0);  
Assert.assertTrue(expected==value);
```

Le test:

```
calc= new EuroCalc(tauxchange);  
double expected=1.0;  
double value=calc.euro2euro(1.0);  
Assert.assertTrue(expected==value);
```

réussit si

- 1 assertTrue réussit
- 2 le comportement de l'objet factice est celui spécifié.

Spécifier le nombre d'appels:

nb-invocation(objet-factice).method(parameters)x

- oneOf: un et un seul appel
- exactly(n).of : exactement n
- atLeast(n): au moins n
- allowing: peut être appelé
- never: aucun appel

La méthode *foo* est exécutée au moins n fois.

- one

Renvoyer des valeurs:

- ...; *will(returnValue(x))*
- ...; *will(onConsecutiveCalls (returnValue(x),
returnValue(y),
returnValue(z)))*

et plus (voir la documentation)