JUnit

Présentation des mocks

Jean-Marc Farinone

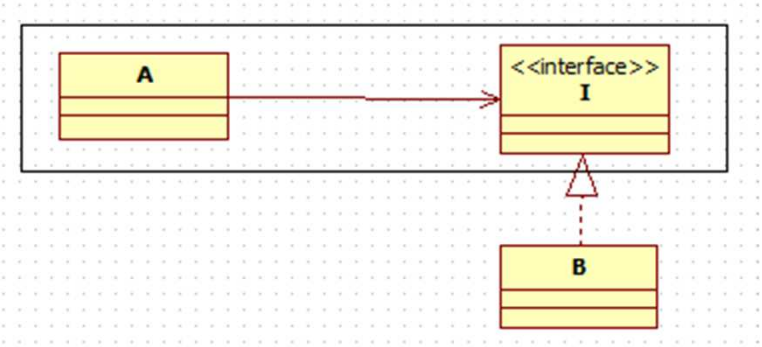
Maître de Conférences
Conservatoire National des Arts et Métiers
CNAM Paris (France)

© JMF (Tous droits réservés)

Rappel ? : inversion de dépendance

- Définition rappel :
  On dit qu'une classe A dépend de classe B si dans la définition de A apparaît l'identificateur B. On note :

- Lorsqu'une classe dépend d'une autre classe, on peut inverser la dépendance en construisant une interface intermédiaire

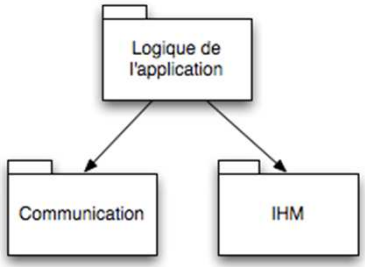


- Désormais c'est la classe B qui dépend du couple (A,I)

© JMF (Tous droits réservés)

Inversion de dépendance : un exemple

- Souvent on commence le développement de sorte que la logique générale de l'application appelle directement les modules de bas niveau :

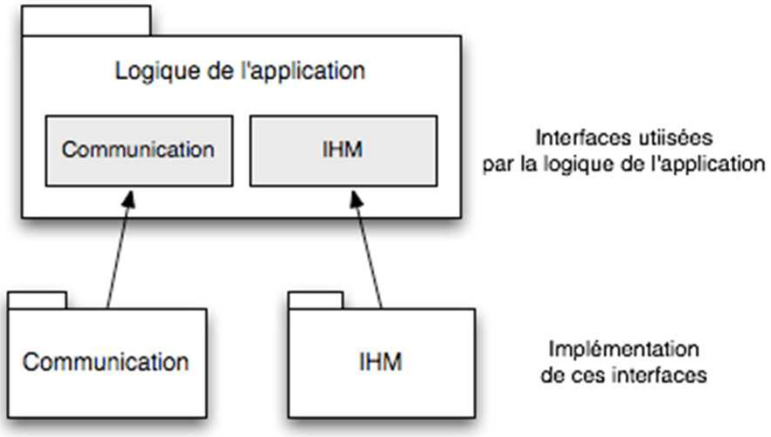


- C'est naturel mais :
- Les modules de haut niveau risque fort de devoir être modifiés lorsque les modules de bas niveau sont modifiés
- Il n'est pas possible de réutiliser les modules de haut niveau indépendamment de ceux de bas niveau. C'est à dire il n'est pas possible de réutiliser la logique d'une application en dehors du contexte technique dans lequel elle a été développée

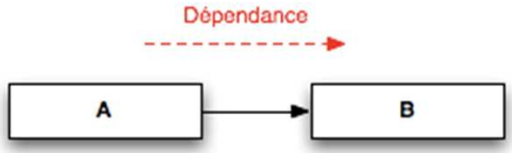
© JMF (Tous droits réservés)

La solution

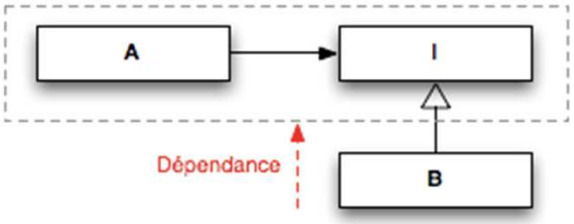
- La bonne solution est de faire en sorte que les modules de bas niveau doivent se conformer à des interfaces définies et utilisées par les modules de haut niveau



- En règle générale, si une classe A dépend d'une classe B,



on peut inverser la dépendance en introduisant une interface qui sera implémentée par la classe B :



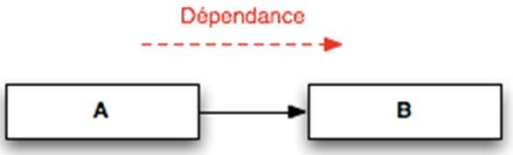
© JMF (Tous droit

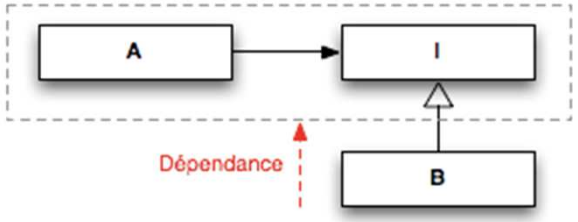
Principe d'inversion des dépendances (DIP)

- Dependency Inversion Principle (DIP)
- Robert C. Martin (~ 1997 C++ Report). *The dependency Inversion Principle*
- "Les modules de haut niveau ne doivent pas dépendre de modules de bas niveau. Tous deux doivent dépendre d'abstractions. Les abstractions ne doivent pas dépendre de détails. Les détails doivent dépendre d'abstractions"

© JMF (Tous droits réservés)

Remarque sur le principe d'inversion des dépendances

- Passer de 

à 

  permet d'inverser la dépendance : B dépend désormais du couple (A,I)

- I forme une sorte d'extension de la classe A dans laquelle on inclut une représentation abstraite des dépendances de A
- La solution est : "dépendre de l'abstraction plutôt que de l'implémentation"

© JMF (Tous droits réservés)

Conclusion sur le principe d'inversion des dépendances

- Isoler les parties génériques/réutilisables de l'application en les faisant reposer uniquement sur des interfaces
- Considérer l'héritage comme une implémentation d'interface, la classe dérivée pouvant se "brancher" dans n'importe quel code qui utilise cette interface (l'interface forme alors un contrat entre le code utilisateur et les classes dérivées)
- Construire les parties "techniques" de l'application sur les parties "fonctionnelles", et non l'inverse

© JMF (Tous droits réservés)

Inversion de contrôle et injection de dépendance

- Plus précisément l'inversion de dépendance est souvent décomposée en deux parties :
- l'inversion de contrôle qui est l'architecture qui montre que le sens des flèches a été inversé
- l'injection de dépendance qui donne une valeur pour une classe qui doit implémenter l'interface intermédiaire

© JMF (Tous droits réservés)

Dépendance entre classes

- Il est très courant qu'une classe contienne des références sur des objets d'autres classes : on a une architecture comme :

```
public class MaClasse {
     private UneAutreClasse ref;
}
```

- Qu'on peut d'ailleurs représenter en UML par :

- Donner des exemples
- On dit que `MaClasse` dépend de `UneAutreClasse` si et seulement si, dans la définition de `MaClasse`, apparaît l'identificateur `UneAutreClasse`
- Mais un test unitaire sur `MaClasse` ne doit faire intervenir que la classe elle même. S'il fait intervenir plusieurs classes entre elles, c'est un test d'intégration et pas un test unitaire !

© JMF (Tous droits réservés)

Mock = simulateur

- Il faut un remplaçant, un simulacre d'objet des classes dont on dépend : ce sont les mocks
- Un "mock" simule une ressource
- Par exemple si une classe est liée à une BD ou à une file JMS ou autre chose, mais qu'on n'a pas (encore) ces "ressources" et qu'on veut quand même tester cette classe, on va utiliser des mocks qui sont des objets simulant ces ressources
- C'est utile aussi pour éviter les erreurs dues au ressources sous-jacentes
- source : `http://www.developpez.net/forums/d120071/java/edi-outils-java/tests-performance/test-junit-mock/`
- Les mocks sont parfois appelés des doublures, des bouchons, des marionnettes, des fantoches, des stubs, ...

© JMF (Tous droits réservés)

Des APIs pour les mocks

- Il existe plusieurs APIs permettant d'utiler les mocks
- Easy Mock :
- site originel : `http://easymock.org/`
- des tutoriaux à : `https://sites.google.com/a/thedevinfo.com/thedevinfo/Home/unit-testing/easymock/easy-mock-tutorial` ou `http://www.slideshare.net/subin123/easymock-tutorial` ou `http://baptiste-wicht.developpez.com/tutoriels/java/tests/mocks/easymock/`
- MockMaker à `http://mockmaker.sourceforge.net/eclipse.html`
- De nombreux autres et ... Mockito

© JMF (Tous droits réservés)

Une API pour les mocks : Mockito

- site originel : `mockito.org`
- Récupérer le `mockito-all-x.x.x.jar` à partir du site original (ou `https://code.google.com/archive/p/mockito/downloads`)
- L'installer dans son projet Eclipse (ou le repérer par classpath)
- Pour manipuler les mocks, on utilise essentiellement des méthodes statiques de la classe `org.mockito.Mockito`

© JMF (Tous droits réservés)

Création de mocks

- Si on a :

```
public class UneAutreClasse {
     public int uneAutreMethode() {
          ... return ...;
     }
}
```

ou

```
public interface MonInterface {
     public int maMethode();
}
```

dont la classe `MaClasse` dépend, on va pouvoir créer des mocks de `UneAutreClasse` ou de `MonInterface` et préciser leur comportement pa

```
import static org.mockito.Mockito.mock;
import org.junit.Test;
public class LeTest {
     @Test
     public void unTest() {
          MonInterface doublure = mock(MonInterface.class);
          ...
          UneAutreClasse ref = mock(UneAutreClasse.class);
          ...
     }
}
```

- Dès qu'un mock est créé (par la méthode statique `org.mockito.Mockito.mock()`), il est immédiatement utilisable et les méthodes qui retournent une valeur, retournent une valeur par défaut (`null`, `0` ou `false`), les autres ont un corps vide

© JMF (Tous droits réservés)

when() et thenReturn()

- Les méthodes statiques `when()` et `thenReturn()` de la classe `org.mockito.Mockito` précise le comportement du mock (et écrase les comportements par défaut)
- Exemple :

```
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

import org.junit.Test;

import dev.MonInterface;
import dev.UneAutreClasse;

public class LeTest {
     @Test
     public void unTest() {
          MonInterface doublure = mock(MonInterface.class);
          when(doublure.maMethode()).thenReturn(56);
          System.out.println(doublure.maMethode());

          UneAutreClasse ref = mock(UneAutreClasse.class);
          when(ref.uneAutreMethode()).thenReturn(87);
          System.out.println(ref.uneAutreMethode());
     }
}
```

- démo `ExempleMockProjet` sous eclipse dans workspace `...\Seance4Mock\Demos`, classe `LeTest1`

© JMF (Tous droits réservés)

Bidouille `mock()`, `when()`

- Pour utiliser les méthodes (statiques) `mock()` et `when()`, Il faut mettre les `import static` :

```
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;
```

- Le faire "à la main" dans Eclipse. CTRL + SHIFT + O ne le fait pas ! (CTRL + SHIFT + O ne fait pas les `import static`)

© JMF (Tous droits réservés)

Pour les méthodes avec argument

- Le corps de `when( )` indique l'appel qui sera fait. Si la méthode à des arguments on l'indique dans le `when( )`
- Par exemple avec l'interface

```
public interface MonInterface {
     public int methodeAvecArgInt(int a);
}
```

on positionne la valeur de retour (18765) de la méthode lorsque son argument a la valeur la valeur 67 par :

```
when(doublure.methodeAvecArgInt(67)).thenReturn(18765) ;
System.out.println(doublure.methodeAvecArgInt(67));
```

© JMF (Tous droits réservés)

Valeur de retour identique pour tous !

- On peut indiquer qu'un méthode retourne la même valeur quelle que soit la valeur de son paramètre à l'aide de `anyInt()`
- Par exemple : `leMock.laMethodeAvecUnArg(anyInt())`
- Par exemple avec l'interface

```
public interface MonInterface {
     public int methodeAvecArgInt(int a);
}
```

on peut créer un mock qui retourne toujours `8765` quelle que soit la valeur du paramètre par :

```
when(doublure.methodeAvecArgInt(anyInt())).thenReturn(8765) ;
System.out.println(doublure.methodeAvecArgInt(54));
System.out.println(doublure.methodeAvecArgInt(9));
```

- Remarque : faire l'
  `import static org.mockito.Mockito.anyInt;`
  pour utiliser `anyInt()`
- source :
  `http://gojko.net/2009/10/23/mockito-in-six-easy-examples/`

© JMF (Tous droits réservés)

Plus sur `when()`, `thenReturn()`

- "Use `when(`*`appel`*`).thenReturn(value)` to specify the stub value for a method. If you specify more than one value, they will be returned in sequence until the last one is used, after which point the last specified value gets returned. (So to have a method return the same value always, just specify it once)
- For example:"

```
package test;

import static org.mockito.Mockito.*;
import static org.junit.Assert.*;
import java.util.Iterator;
import org.junit.Test;

public class MonTest {
     @Test
     public void iterator_will_return_hello_world(){
          Iterator i = mock(Iterator.class);
          when(i.next()).thenReturn("Hello").thenReturn("World");

          String result= "" + i.next();
          result += " "+i.next();

          System.out.println(result);
          assertEquals("Hello World", result);
     }
}
```

- source : `http://gojko.net/2009/10/23/mockito-in-six-easy-examples/`

© JMF (Tous droits réservés)

Appels successifs avec `when()`, `thenReturn()`

- L'appel `when(`*`appel`*`).thenReturn(value).thenReturn(value2);` permet d'indiquer que le premier `appel` retournera `value` et le second appel `value2`
- Une autre syntaxe possible est `when(`*`appel`*`).thenReturn(value, value2)`

```
//when(doublure.maMethode()).thenReturn(3, 4) ;
// est un raccourci pour
when(doublure.maMethode()).thenReturn(3).thenReturn(4);
System.out.println(doublure.maMethode());
System.out.println(doublure.maMethode());
```

- Par la suite les appels suivants retournent `value2`
- Démo ExempleMockProjet sous eclipse dans workspace `...\Seance4Mock\Demos`, classe `LeTest2`

© JMF (Tous droits réservés)

Utilisation d'un mock

- Finalement un mock s'utilise ainsi :
- On le crée (par `mock(...)`)
- On précise, pour chaque appel de méthode, la valeur que cette méthode doit retourner
- On utilise les méthodes ainsi "initialisées"
- C'est simple : normal c'est un mock !
- Par la suite, le code du test fait intervenir les méthodes de la classe à tester et ces méthodes retournent ce qui est prévu : c'est le testeur qui l'a écrit !
- Bref, le code des méthodes des classes autre que la classe à tester sont mis dans les mocks

© JMF (Tous droits réservés)

Bibliographie sur les mocks

- Une présentation des mocks à `http://www.ibm.com/developerworks/library/j-mocktest/index.html`
- Une présentation des mocks (en français) `http://fr.wikibooks.org/wiki/Introduction_au_test_logiciel/Doublures_de_test`
- JUnit in Action, Petar Tahchiev et al., ISBN 1-930110-99-5; ed Hanning

© JMF (Tous droits réservés)

Bibliographie sur Mockito

- Le site originel de Mockito : `mockito.org`)
- 6 exemples pour Mockito à `http://gojko.net/2009/10/23/mockito-in-six-easy-examples/`. Cet article a fortement inspiré cet exposé
- Un exemple pour Mockito `http://schuchert.wikispaces.com/Mockito.LoginServiceExample`

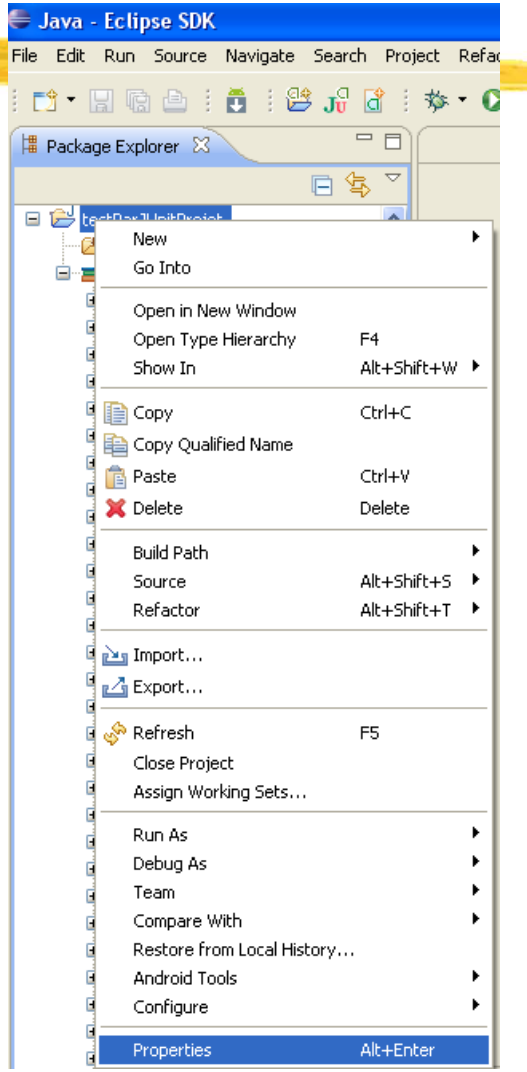
© JMF (Tous droits réservés)

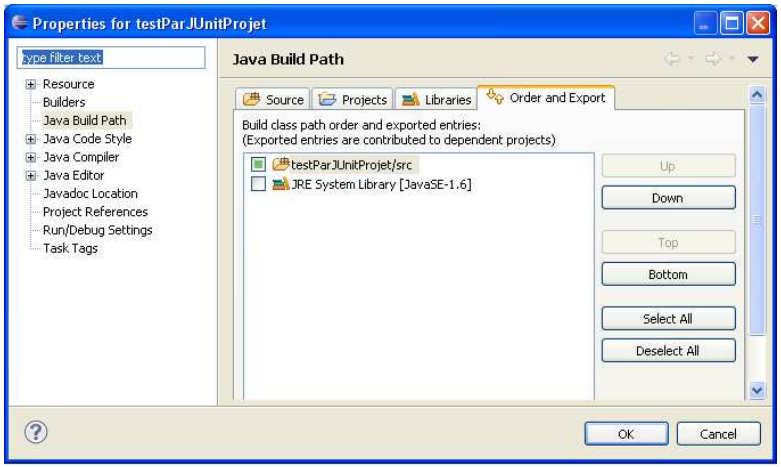
Compléments
Ajout d'une bibliothèque dans Eclipse

© JMF (Tous droits réservés)

Ajouter les `.jar` de tests au projet (1/4)

- Il faut d'abord, si vous ne l'avez pas fait par ailleurs, ajouter les `.jar` utiles comme bibliothèque de classes additionnelle à votre projet
- Pour cela, sélectionner le projet, cliquez droit, puis cliquez Properties. Sélectionner Java Build Path. Apparaît la fenêtre :

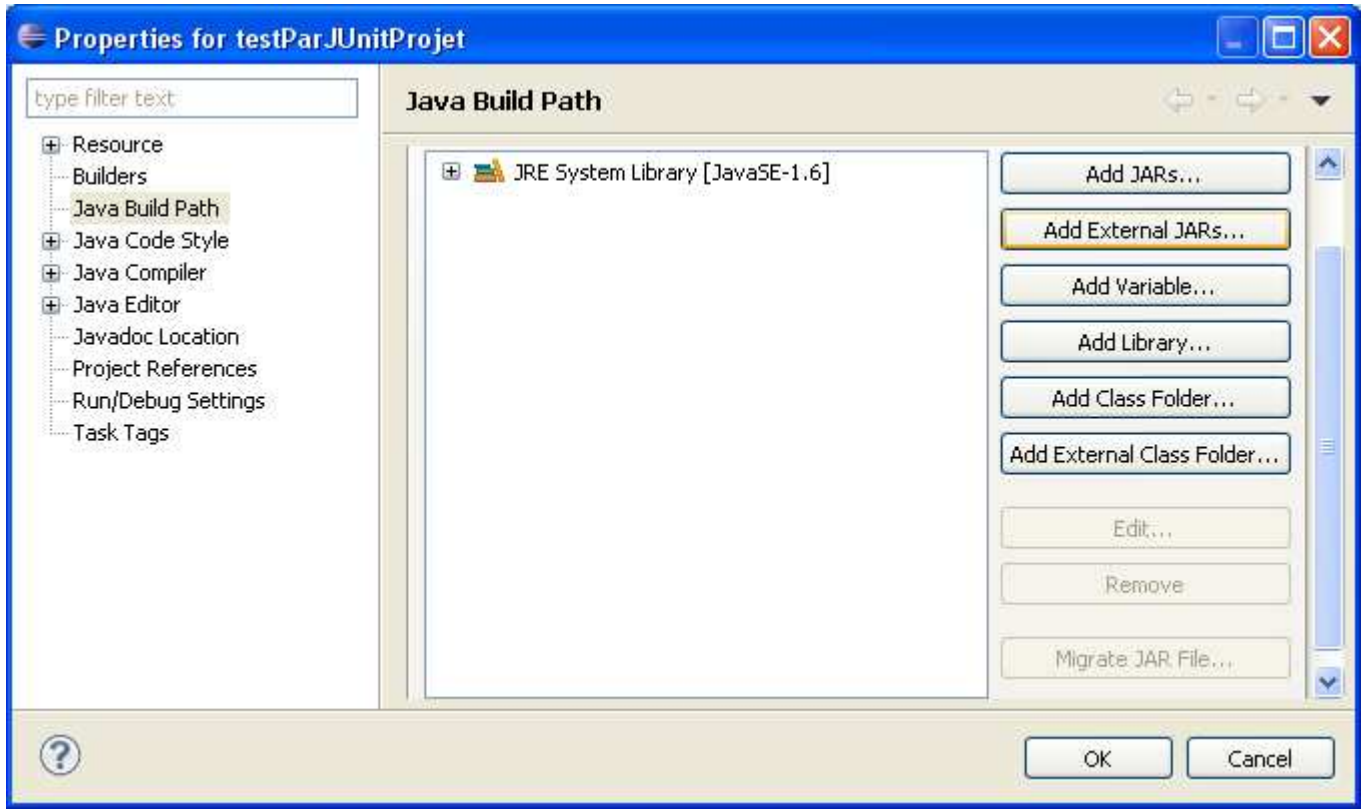




© JMF (Tous droits réservés)

Ajouter les `.jar` de tests au projet (2/4)

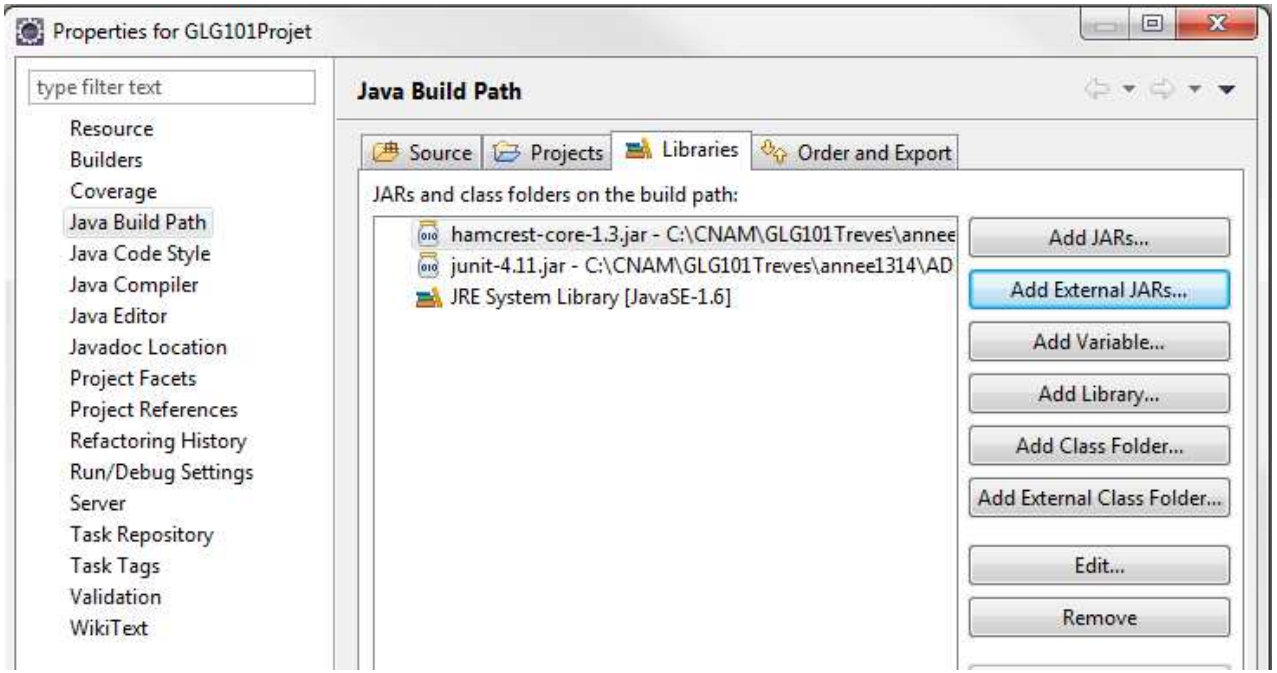
- Sélectionnez l'onglet "Libraries",
- Cliquez le bouton "Add External JARs..."



© JMF (Tous droits réservés)

Ajouter les `.jar` de tests au projet (3/4)

- Parcourez votre système de fichiers pour trouver les `.jar` (`junitXXX.jar` et `hamcrest-core-XXX.jar`)
- Après l'avoir sélectionné, ils apparaissent comme bibliothèques supplémentaires :

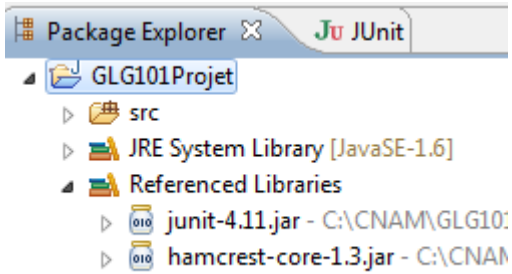


- C'est fini !

© JMF (Tous droits réservés)

Ajouter les `.jar` de tests au projet (4/4)

- Vérification : Dans "Referenced Libraries" du "Package Explorer" apparaît le `junitXXX.jar`



© JMF (Tous droits réservés)

Fin

© JMF (Tous droits réservés)