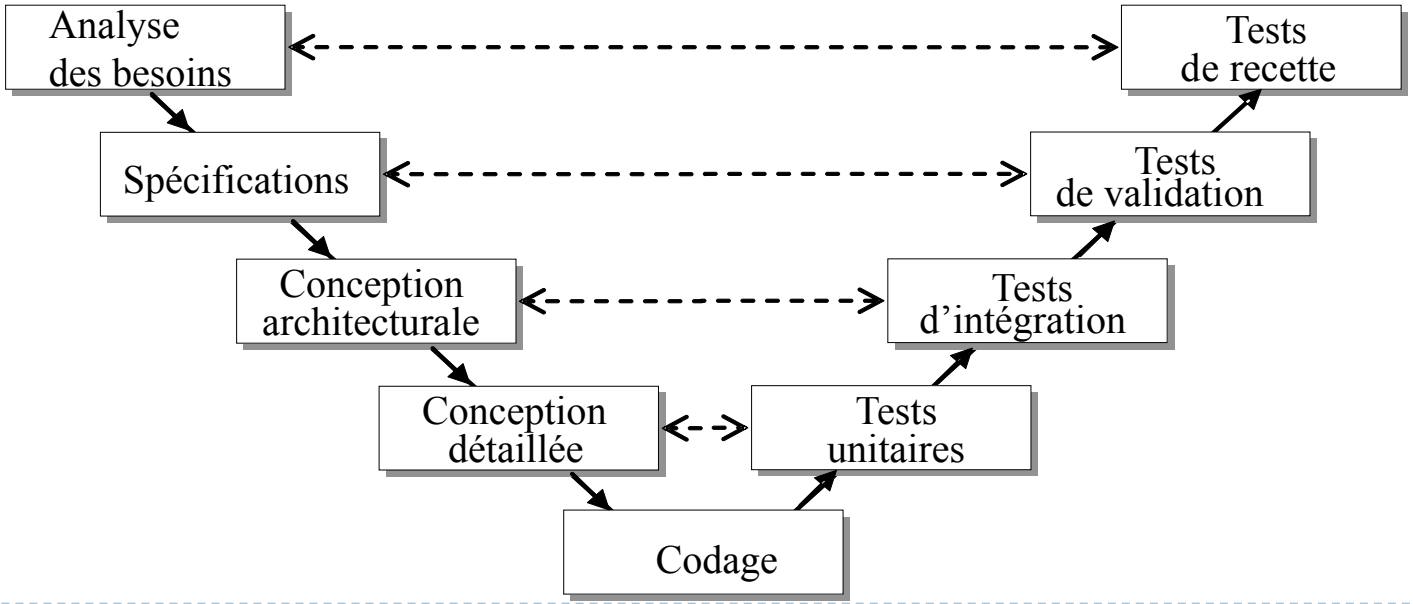
# CM5 – Testabilité & Doublure

Gerson Sunyé — gerson.sunye@univ-nantes.fr
Mottu - Sunyé

# Introduction à la Testabilité

- Objectif
  - Prendre conscience qu'il ne suffit pas de décider de tester pour
    - Tester facilement
    - Trouver toutes les erreurs



# Introduction à la Testabilité

- Tester
  - Construire un ensemble de cas de test
  - Cas de test
    - Une description
    - Une initialisation
      - Par exemple créer un objet, le mettre dans un état précis
    - Une donnée de test
      - Par exemple certains paramètres de la méthode à tester
        (on détermine ces paramètres avec les techniques des précédents cours)
    - Un oracle
      - Contrôler que l'exécution de la donnée de test respecte la spécification

# 2 principales heuristiques de la Testabilité

- Observabilité
  - Qu'est-ce qu'il est possible d'observer dans le logiciel ?
    - Peut-on observer une seule/toutes les partie(s) du logiciel ?
      - Classes, propriétés, variables
    - Peut-on observer l'environnement ?
      - Le temps qui s'écoule, l'interaction avec d'autres logiciels
- Contrôlabilité
  - Qu'est-ce qu'il est possible de manipuler ?
    - Logiciel sous test, l'environnement
  - Peut-on mettre le système dans l'état voulu ?
    - Faire des tests sur un système en marche ?

# Heuristiques secondaires de la Testabilité

- Disponibilité
  - Logiciel disponible (boite noire et blanche)
  - Le logiciel doit être suffisamment développé pour exécuter les tests
  - La spécification doit être explicite et disponible
- Stabilité
  - Les éléments testés sont modifiés en fournissant un suivi

# Testabilité

- La testabilité s'évalue et complète les tests

- La testabilité est l'activité qui permet d'évaluer la capacité à tester un système selon plusieurs points de vue:
  - La capacité d'un logiciel à être testé
    - (observabilité, contrôlabilité, disponibilité, stabilité)
  - La capacité des tests à faire des vérifications pertinentes
    - (observabilité, contrôlabilité, stabilité)
  - La capacité d'une spécification à formuler des exigences vérifiables
    - (disponibilité, stabilité, observabilité)

# Testabilité du point de vue conception logicielle

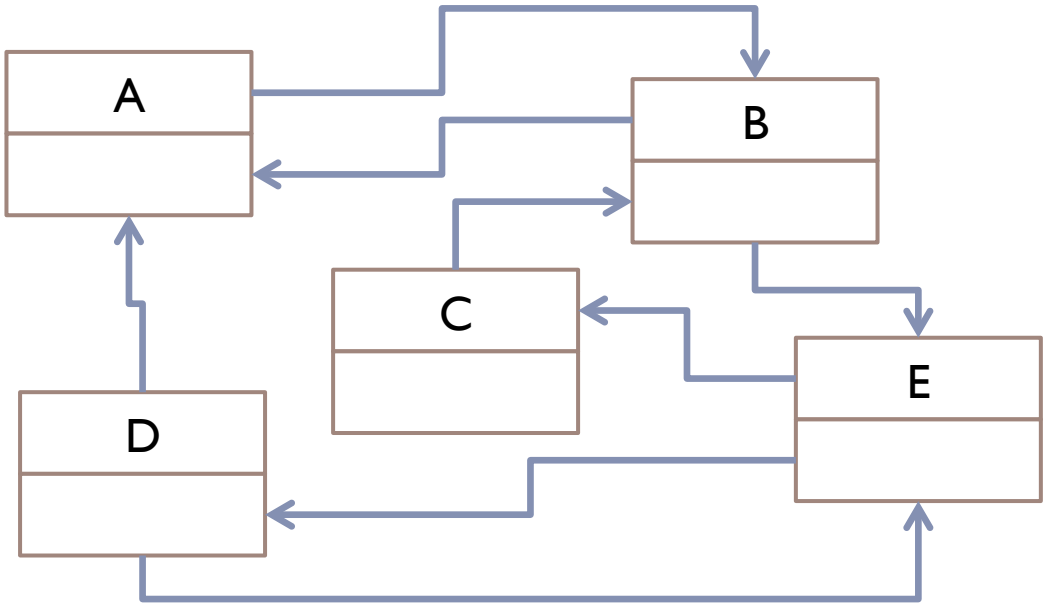
- Observabilité
  - Pour la génération de données de test
    - Boite blanche/noire
    - Interface
  - Pour l'oracle
    - Qu'est-ce que l'oracle va observer pour détecter un comportement incorrecte ?
    - Découpage des classes, des méthodes (statiques ?)
  - Pour le test d'évolutions du logiciel et son diagnostic
    - Trace d'exécution

# Testabilité du point de vue design du logiciel

- Contrôlabilité
  - Pour l'initialisation d'un cas de test
    - Peut-on créer un objet indépendemment du reste du système ?
    - Peut-on contrôler le logiciel sous test sans le corrompre ?
      - Pendant son fonctionnement
  - Pour l'oracle
    - Est-ce que les accesseurs utilisés ont des effets de bords ?

# Testabilité du point de vue design du logiciel

- Exemples de choix du design pouvant être préjudiciables à la testabilité
  - Les variables privées d'une classe ne sont pas observables
  - Peu de typage, typage dynamique
  - Les interdépendances entre classes provoquent des interblocages



# Evaluer la testabilité du logiciel

- Typage ? Typage dynamique ?
- Mesure
  - Taille des classes, des méthodes, etc.
  - Sera vu dans un prochain cours

# Mesurer la testabilité des tests vis-à-vis du logiciel

- Mesurer la capacité des tests à détecter des erreurs
  - Technique de l' « Analyse de mutation »
    - Tests versus les erreurs potentiels du logiciel
    - Prochain cours

- Critères de test
  - Fonctionnel -> partition des domaines des variables d'entrée/sortie
  - Structurel -> couverture du graphe de flot de contrôle

  - Empirique, dynamique -> couverture du code testé
    - Est-ce que toutes les instructions ont été exécutées ?
      - Conséquence directe de la couverture du graphe de flot de contrôle
    - Est-ce que toutes les données ont été employées ?

# La testabilité permet de

- Améliorer les tests
  - S'il faut davantage de tests
  - Certaines parties du logiciel doivent être davantage/ différemment testées
- Améliorer la conception du logiciel
  - Permettre l'observation de son état et de son comportement
  - Pas de complexité superflue
  - Plus de typage, moins d'interdépendance, etc.
- Améliorer la formalisation de la spécification
- Connaitre les limites des vérifications effectuées
  - Le test n'est pas exhaustif
  - Il faut savoir quels risques restent après le test

# Comment anticiper/résoudre les problèmes de testabilité ?

- Dès la spécification, conception
  - Considérer les besoins de contrôlabilité, d'observabilité
- Comment faire quand on ne peut pas contrôler certaines parties du code ?
  - Le temps
  - La communication avec l'extérieur du programme
    - D'autres programmes
    - Des liaisons vers le monde physique, nettement moins contrôlable
      - Contrôle impossible
      - Contrôle possible mais avec des coûts/temps non raisonnables

# Améliorer la testabilité dès la conception

- Améliorer la structure des packages
  - Diminuer les interdépendances entre classes.

- Limiter la complexité des classes et des méthodes
  - De trop nombreuses imbrications de boucles et de conditionnelles font exploser la combinatoire pour résoudre la sensibilisation des chemins du graphe de flot de contrôle
    - En diminuant le nombre de chemin d'une méthode, on augmente sa testabilité [Nejmeh 1988]
  - L'analyse du flot de données est aussi important
    - Rapport entre la définition et l'utilisation des variables

# Améliorer la testabilité dès la conception

- Permettre l'observation intermédiaire d'état
  - Un programme dont l'état interne est important est moins testable qu'un autre n'ayant que des entrées et des sorties
  - L'état interne est-il masqué ?
  - L'état interne est-il observable en continue ?
    - On peut ajouter des observateurs
    - Des contraintes embarquées
      - À voir au prochain cours
  - Important pour le test mais aussi la localisation d'erreur

# Interdépendance limite la testabilité

Doublure de test

# Isolation

- Comment tester efficacement une unité qui dépend d'une autre qui:
  - n'existe pas encore.
  - n'est pas fiable.
    - e.g. impliqué dans des cycles d'interdépendances
  - est très lente.
  - est très coûteuse

# Solution

- Remplacer l'unité dont dépend l'unité sous test par un équivalent spécifique au test.

# Principe

- Remplacement de l'unité dont dépend l'unité sous test par un équivalent, la doublure de test.
- Différentes variantes de doublures sont possibles.
- Indépendamment de la variante, seul le comportement attendu par un test est implémenté.

# Utilisation

- Code difficile à tester.
- Tests dont la mise en place est coûteuse.
- La configuration est complexe.
- Des effets de bord possibles.
- La mise en place de conditions exceptionnelles est complexe.

# Exemple: test de l'accès à une base de données

# Cas de test: étapes

1. Configuration.
2. Exécution.
3. Oracle.
4. Nettoyage.

# Exemple : Problèmes de base de données

- Répétabilité:
  - Etat de la base de données lors d'une erreur ?
  - Intégrité référentielle, déclencheurs (triggers), etc.
- Situations exceptionnelles:
  - Timeouts, errors, etc.

# Difficultés

- La configuration est complexe.
- Des effets de bord possibles.
- La mise en place de conditions exceptionnelles est complexe.

# Cas de test: étapes

1. Configuration.
2. Exécution.
3. Oracle.
4. Nettoyage.

# Exemple

1. Configuration.
2. Exécution.
3. Oracle.
4. Nettoyage.

```
INSERT INTO Client (Name, Date)
VALUES ('Clémentine', '20-Jan-1986');
INSERT INTO Client (Name, Date)
VALUES ('Myrtille', '2-May-1987');

-- Assertions

DELETE * FROM Client
```

# Problèmes

- Répétabilité:
  - Etat de la base de données lors d'une erreur ?
  - Intégrité référentielle, déclencheurs (triggers), etc.
- Situations exceptionnelles:
  - Timeouts, erreurs, etc.

# Alternatives

- Données de production (réelles).
- Duplication de données
- Données de test indépendantes.
- Doublures.

# Données de production

- Répétabilité difficile, spécialement si les tests échouent.
- Difficile de tester les conditions exceptionnelles.
- Difficile d'isoler les tests.
- Données sensibles (comptes, informations confidentielles, etc.)

# Duplication

- Répétabilité difficile, spécialement si les tests échouent.
- L'information est dupliquée.
- Changements (e.g. schéma) doivent être faits au moins 2 fois.
- Parfois, les différences sont plus importantes que prévu: sécurité, triggers, procédures stockées, etc.
- Difficile de tester les conditions exceptionnelles
- Difficile d'isoler les tests.

# Données de test indépendantes

- Répétabilité difficile, spécialement si les tests échouent.
- Difficile de tester les conditions exceptionnelles.
- Difficile d'isoler les tests.

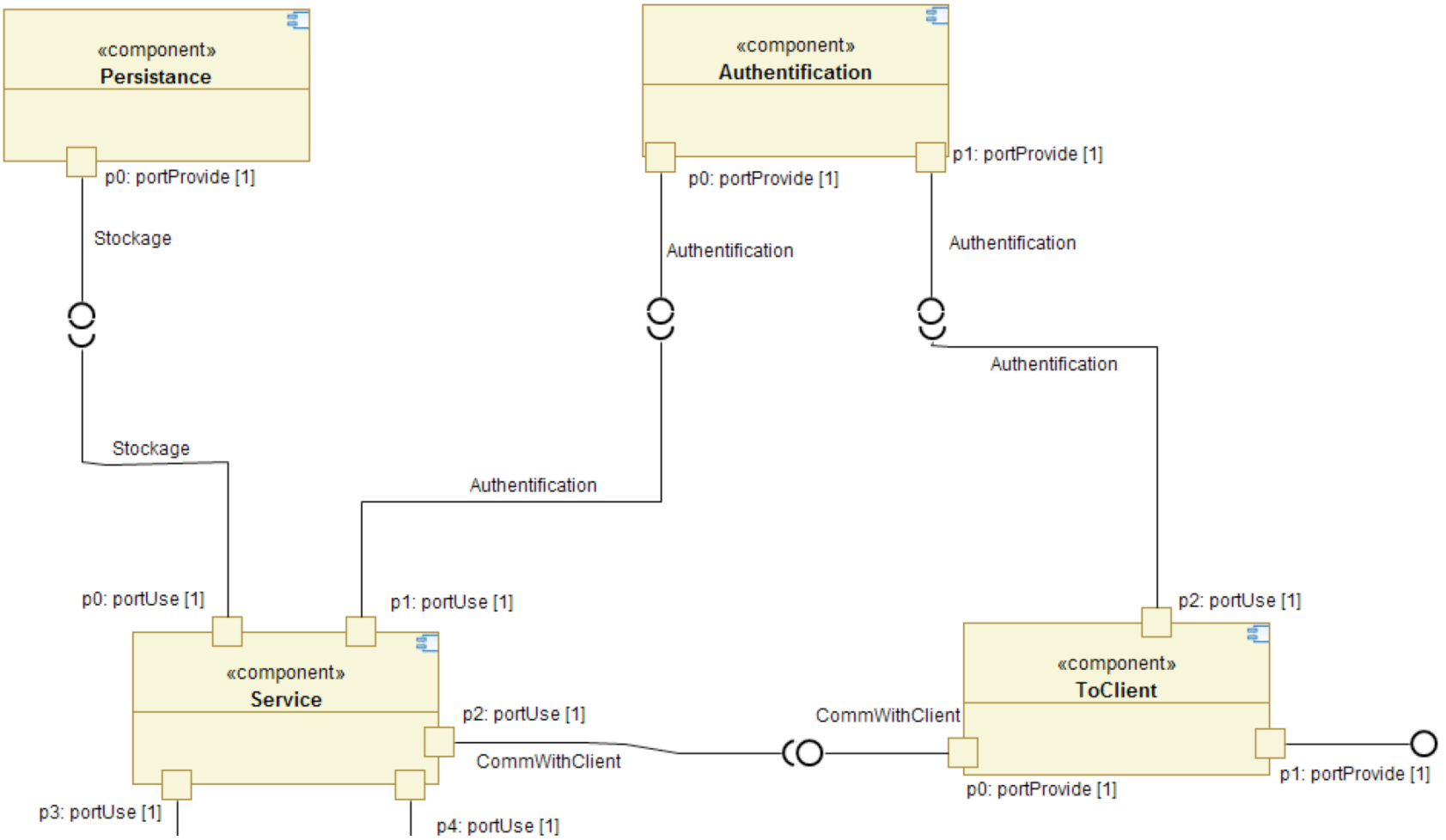
# Doublures

- Utilisation d'une doublure de test capable de remplacer l'accès à la base de données.
- Alternatives:
  - Créer des doublures des objets d'accès à la base de données.
  - Le faire à un plus haut niveau: composants, couche de persistance, etc.

# Doublure d'objets d'accès

- En jdbc: Connection, Statement, ResultSet, etc.
- Adaptée au test d'une seule méthode.
- Moins adaptée aux systèmes contenant des requêtes multiples (environ 3 doublures par requête).

# Doublure de composants



# Bénéfices

- Réduction de la dépendance lors des tests unitaires.
- Découplage.
- Test de situations exceptionnelles.
- Répétabilité.
- Amélioration de la conception.

# Autres exemples

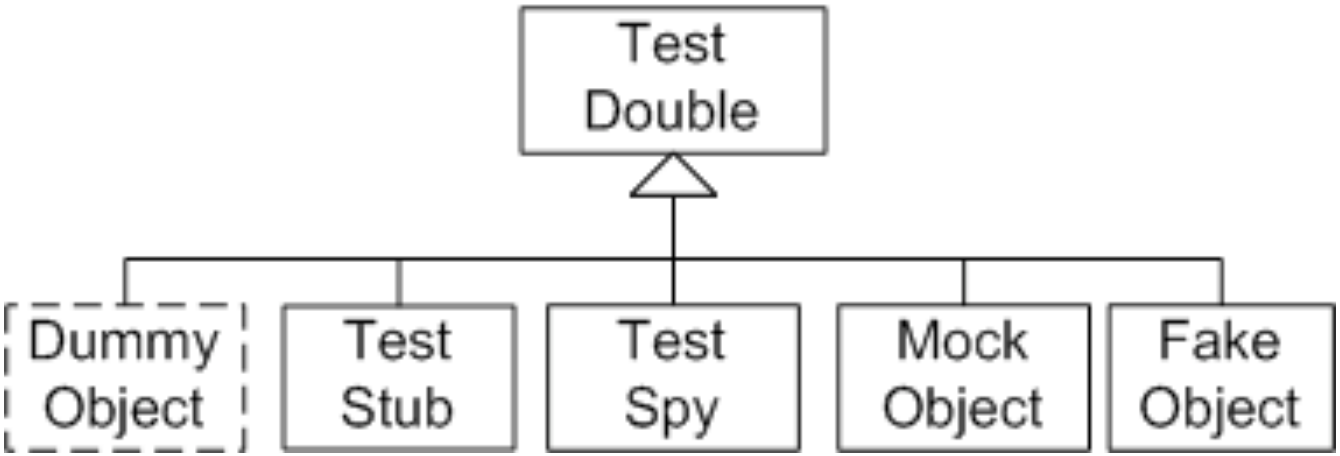
- IHM
- JMS
- JSP/Servlets
- Accès fichier.

# Exemple : Doublure de base de données

- Utilisation d'une doublure de test capable de remplacer l'accès à la base de données.
- Alternatives:
  - Créer des doublures des objets d'accès à la base de données.
  - Le faire à un plus haut niveau: composants, couche de persistance, etc.

# Types de doublures

# Doublures de test [Meszaros]




http://xunitpatterns.com/Test%20Double.html

# Doublures de test

| en | fr | description |
|---|---|---|
| Stub | Bouchon | Classe codée à la main. Fournit des réponses pré-définies aux appels faits durant le test. |
| Mock | Simulacre | Les attentes sont adaptées avant l'exécution de la méthode de test. |
| Spy | Espion | La vérification se fait après l'exécution de la méthode de test. |

# Doublures de test

| en | fr | description |
|---|---|---|
| Dummy | Fantôme | Objets vides, utilisés simplement pour remplir des listes de paramètres. |
| Fake | Substitut | Objets qui implémentent le même comportement que l'original, mais de façon plus simple. |

# Bouchons de test pour la simulation de code incontrôlable

- Les Mocks sont des Stubs paramétrés depuis les cas de test (Junit par exemple), schématiquement :
  - Les stubs remplacent des classes testées pour casser les interdépendances
  - Les mocks remplacent des classes dont on n'a pas le contrôle
- Les Mocks permettent de résoudre principalement des problèmes de testabilité niveau « contrôlabilité »

# Mock

- Un Mock est un Stub paramétré depuis les cas de test (Junit par exemple)
  - Spécifie les valeurs retournées quand le stub est appelé
  - Spécifie les appels attendus depuis telle classe vers le stub
  - Spécifie combien de fois les méthodes « stubbées » doivent être appelées

# Mock et test d'intégration

- Facilite la substitution d'une classe par un seul bouchon paramétrable
  - Moins on perturbe l'architecture moins on risque les erreurs de test

# Mock et classes non contrôlables

- Certaines classes ne sont pas contrôlables
  - Externes au système testé
  - Basées sur le temps, le monde physique
- Certaines classes ne peuvent pas être suffisamment contrôlées
  - Leurs fonctions principales le sont
  - Pas les critères extra-fonctionnels
    - Temps d'exécution
    - Consommation  mémoire
    - etc.
- Mock permettent de tester le système en simulant au besoin

# Outillage de Mock

- Mockito
- EasyMock
- Jmockiyt

# Mockito

# Principes (1/2)

- Générateur de doublures.
- Plus précisément, de classes-espionnes.

# Principes (2/2)

- Etapes d'utilisation:
  1. Création des doublures.
  2. Description du comportement attendu.
  3. Utilisation des doublures.
  4. Vérification que l'interaction avec les doublures est correcte.

# Exemple

- Création de la doublure grâce à la méthode mock().
- Description du comportement attendu grâce à la méthode when().
- Utilisation de la doublure.
- Vérification.

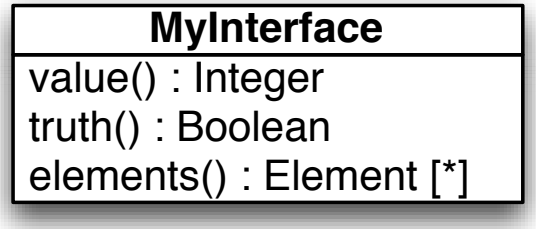
```
import static org.mockito.Mockito.*;
MonInterface doublure =
mock(MonInterface.class);
when(doublure.foo()).thenReturn(42);

doublure.foo();
verify(doublure).foo();
```

# La méthode mock() (1/2)

- Signature:
- `public static <T> T mock(java.lang.Class<T> classToMock)`
- `public static <T> T mock(java.lang.Class<T> classToMock, String name)`
- Permet la création de doublures de classes et d'interfaces.

# La méthode mock() (1/2)

- Lors de la création de la doublure, un comportement par défaut est défini:
- `MyInterface mock = mock(MyInterface.class, «myclass»);`
- `assert «myClass».equals(mock.toString());`
- `assert mock.value() == 0;`
- `assert !mock.truth();`



# La méthode when()

- Méthode utilisée pour associer un comportement à une méthode:
- Associée aux méthodes thenReturn() et thenThrow():
- `when(mock.value()).thenReturn(42);`

# La méthode when()
## Retour de valeurs uniques et consécutives.

- Il est possible de retourner des valeurs uniques et des valeurs consécutives:

```
when(mock.value()).thenReturn(42);
assert mock.value() == 42;
assert mock.value() == 42;
assert mock.value() == 42;

when(mock.value()).thenReturn(4
2,44,99);
assert mock.value() == 42;
assert mock.value() == 44;
assert mock.value() == 99;
assert mock.value() == 99;
```

# La méthode when()
# Retour en fonction des arguments.

- Il est possible de définir la valeur de retour en fonction des valeurs passées en paramètre:

```
when(mock.foo(4,2)).thenReturn(42);
when(mock.foo(1,1)).thenReturn(11);
when(mock.answer(anyInt()).thenReturn(42);

assert mock.foo(1,1) == 11;
assert mock.foo(4,2) == 42;
assert mock.answer(999) == 42;
assert mock.answer(137) == 42;
```

# La méthode when()
## Levée d'exceptions.

- Le comportement associé à une méthode peut être aussi une Exception:

```
when(mock.foo(4,2)).thenReturn(42);
when(mock.foo(1,1)).thenThrow(new
RuntimeException());

assert mock.foo(4,2) == 42;
try {
    mock.foo(1,1);
    assert false;
} catch(RuntimeException e) {
    assert true;
}
```

# La méthode verify()

- Les interactions réalisées avec une doublure sont enregistrées et peuvent être vérifiées à posteriori.

```
verify(mock).value() // appelée exactement 1
fois
verify(mock,times(1)).value() // aussi
verify(mock, atLeastOnce()).value(); // au
moins
verify(mock, atLeast(3)).value(); // au
moins
verify(mock, atMost(6)).value(); // au plus
verify(mock, never()).value(); // jamais
```

# La méthode verify()
## Vérification de l'ordre chronologique.

- La classe InOrder permet de vérifier l'ordre des appels des méthodes sur une ou plusieurs doublures:

```
import org.mockito.InOrder;
InOrder inOrder = inOrder(mock);
inOrder.verify(mock).foo(4, 2); // foo(4,2) a été
inOrder.verify(mock).foo(1, 1); // appelée avant
foo(1,1)
InOrder inOrder = inOrder(mock, mockBis);
inOrder.verify(mock).foo();
inOrder.verify(mockBis).fooBis();
```

# La méthode verify()
## Vérification de l'absence d'interactions.

- Il est possible de vérifier qu'un appel n'a pas été réalisé:

```
mock.foo(4,2);
verify(mock).foo(4,2); // vérification simple
verify(mock, never()).foo(3,9); // v. appel
jamais réalisé
verifyZeroInteractions(mockBis, mockTer);
// vérification de que les autres objets
// n'ont pas eu d'interaction.
```

# La méthode verify()
## Vérification de l'absence de redondance.

- Il est possible de vérifier que seulement certaines interactions ont été réalisées:

```
mock.value();verify(mock).value(); //
vérification simple assert
verifyNoMoreIntercations(mock).
```

# Limites

- Il n'est pas possible de faire de doublures de:
  - Classes finales.
  - Méthodes finales.
  - Méthodes de classe (statiques).
  - Méthodes equals() et hashCode(), utilisées par Mockito.

# Conclusion

# Autres utilisations

- Garantie d'un ordre précis d'appels.
- Aider à l'amélioration de la couverture de code.
- Garantie de que le code de nettoyage est bien appelé.
- Aide au développement initial (on commence avec les doublures avant d'implémenter les vrais objets).

# Bénéfices des doublures

- Les modifications des tests sont plus simples.
- Restreint l'accès aux seules méthodes disponibles.
- Améliorent indirectement la conception.
- Configuration plus simple de situations exceptionnelles.

# Problèmes

- Coût de développement:
  - Courbe d'apprentissage.
  - Programmation par interfaces
- Coût de maintenance:
  - La doublure doit rester synchronisée avec l'objet réel.

# Quand les utiliser

- Pour réduire les dépendances.
- Quand un certain comportement est difficilement reproductible (exceptions).
- Quand un composant n'est pas encore disponible.
- Pour réduire la complexité de la configuration du test.

# Références


- *xUnit Test Patterns: Refactoring Test Code*. Gerard Meszaros. Addison-Wesley.
- http://xunitpatterns.com/
- http://www.mockobjects.com/