Test logiciel et Mocks

Clémentine Nebut

LIRMM / Université de Montpellier

Septembre 2020

Sommaire

Sommaire

Le test

Principe

Essayer pour voir si ça marche ...

Essayer ...

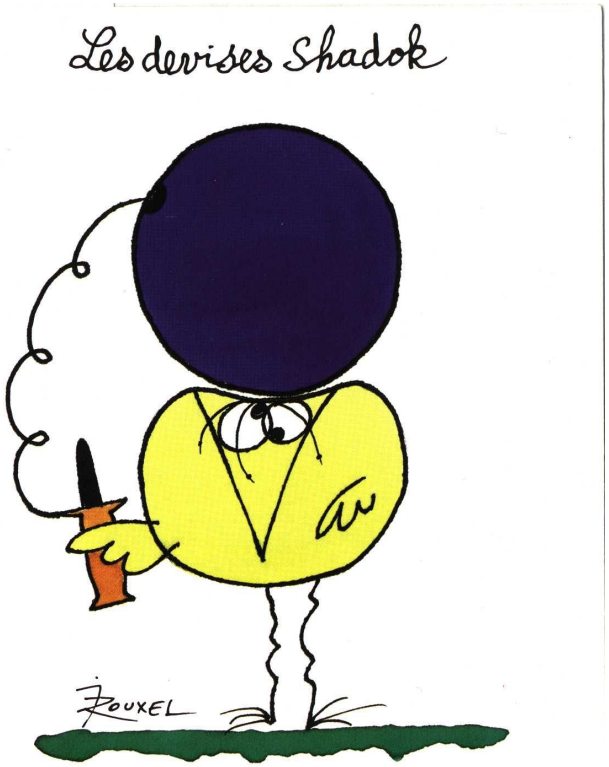
- Comment ça marche ?
  - Démarrage du programme ?
  - Interface graphique ? Textuelle ?
  - ça marche comment une API ?
- Quelles entrées ?
  - Données requises ?
- Qu'est-il possible de faire ?
  - Si on veut tout essayer, il faut savoir ce qu'il y a à essayer !
  - Quels enchaînements nécessaires pour essayer une fonctionnalité ?

... pour voir ...

- Que peut-on voir ?
  - une couleur dans une interface graphique ?
  - un affichage dans une fenêtre ?
  - la valeur d'une variable ?
  - le résultat d'un calcul intermédiaire ?
- Notion d'observabilité

... si ça marche.

- Comment sait-on que ça marche ?
  - au fait, il doit faire quoi ce programme ?
    - notion de spécifications
  - à partir de ce que l'on peut voir, déterminer si ça marche
    - et si on ne voit pas ce que l'on veut ?
- Et si ça ne marche pas ?
  - Diagnostique
- Et si ça a l'air de marcher ...
  - est-on sûr que ça marche vraiment ?
    - notion de confiance $\neq$certitude
  - et si c'étaient les tests qui étaient mauvais ou insuffisants ?
    - qualité des tests, critère d'arrêt


Les devises Shadok
ROUXEL
EN ESSAYANT CONTINUELLEMENT
ON FINIT PAR RÉUSSIR. DONC:
PLUS ÇA RATE, PLUS ON A
DE CHANCES QUE ÇA MARCHE.

Vers une définition ...

Définition de Myers, 1979

Testing is the process of executing a program with the intent of finding errors. [G. Myers. The Art of Software Testing. 1979]

- Reste à savoir ce qu'on teste et ce qu'est une erreur ...

Qu'est ce qu'on teste ?

(quelles propriétés)

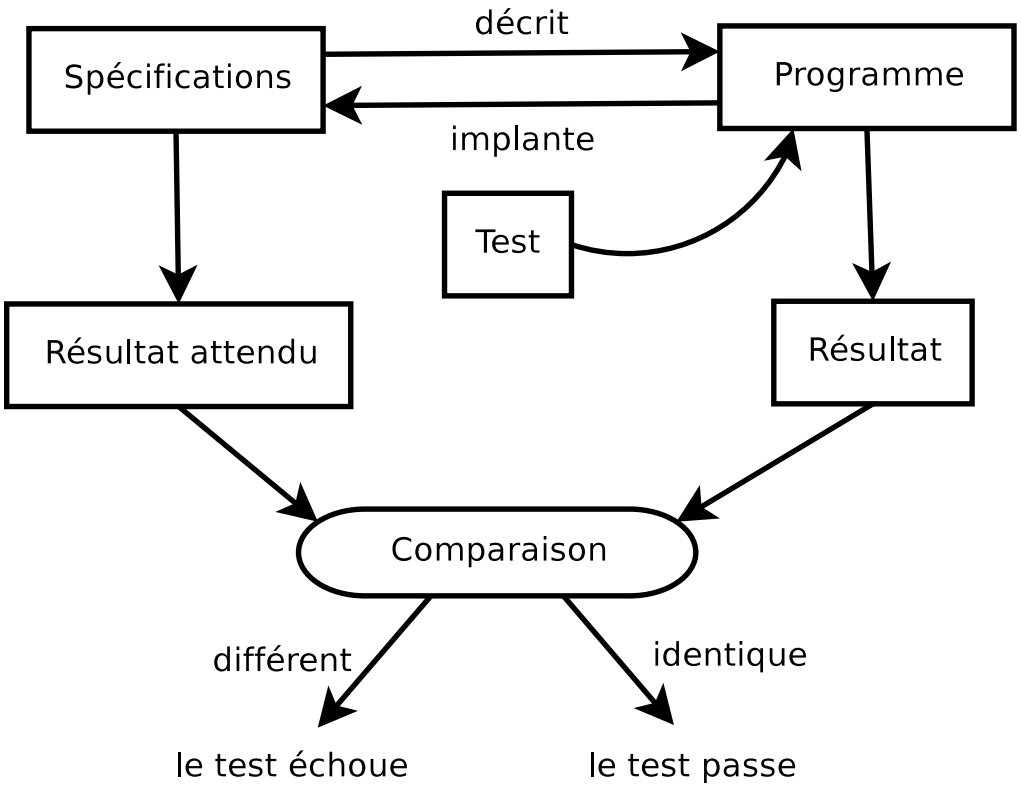
Différentes propriétés à tester

- satisfaction des fonctionnalités requises
- qualité de service (temps de réponse, utilisation mémoire, ...)
- robustesse
- sûreté de fonctionnement
- utilisabilité

On teste vis à vis d'une spécification !

Pour déterminer si on a détecté un problème, toutes les propriétés doivent être spécifiées.

Le verdict ...


décrit
Spécifications
Programme
implante
Test
Résultat attendu
Résultat
Comparaison
différent
identique
le test échoue
le test passe

Quelques principes ...

1. Le programmeur ne doit pas être le testeur ...
2. Ne pas faire les tests en prenant comme hypothèse qu'il n'y a pas d'erreur ... être suspicieux
3. La définition des sorties doit être effectuée AVANT l'exécution du test et pas après ...
4. Faire une analyse soigneuse des traces/résultats
5. Faire des jeux de test avec des entrées valides ET invalides
6. Tester que le logiciel fait ce qu'il doit faire ... mais aussi regarder ce qu'il se passe quand on lui fait faire ce qu'il ne doit pas faire (robustesse)

Des alternatives ?

Le test et les méthodes formelles

- On fait des spécifications formelles, puis :
  - on dérive automatiquement le code
  - ou on écrit un code et on le prouve correct

Des alternatives ?

Le test et les méthodes formelles

- Mais des problèmes de fond ...
  - adéquation entre les spécifications et le cahier des charges ?
  - on prouve des propriétés, et si on a oublié des propriétés à prouver ? (problème identique avec le test)
- Et des problèmes plus terre à terre ...
  - formation d'experts en méthodes formelles
  - coût de mise en place
  - peu de robustesse au changement dans les spécifications

Test et V&V

- V&V = Vérification et Validation
- Le test est une technique particulière de V&V
- Si le test est indispensable dans la quasi-totalité des projets, il n'est pas l'unique procédé de V&V à mettre en œuvre. Par exemple :
  - revues techniques
  - analyseurs statiques

Sommaire

Différents tests

Plusieurs niveaux (échelles)

- Unitaire
- Intégration
- Système
- Acceptation (ou recette)

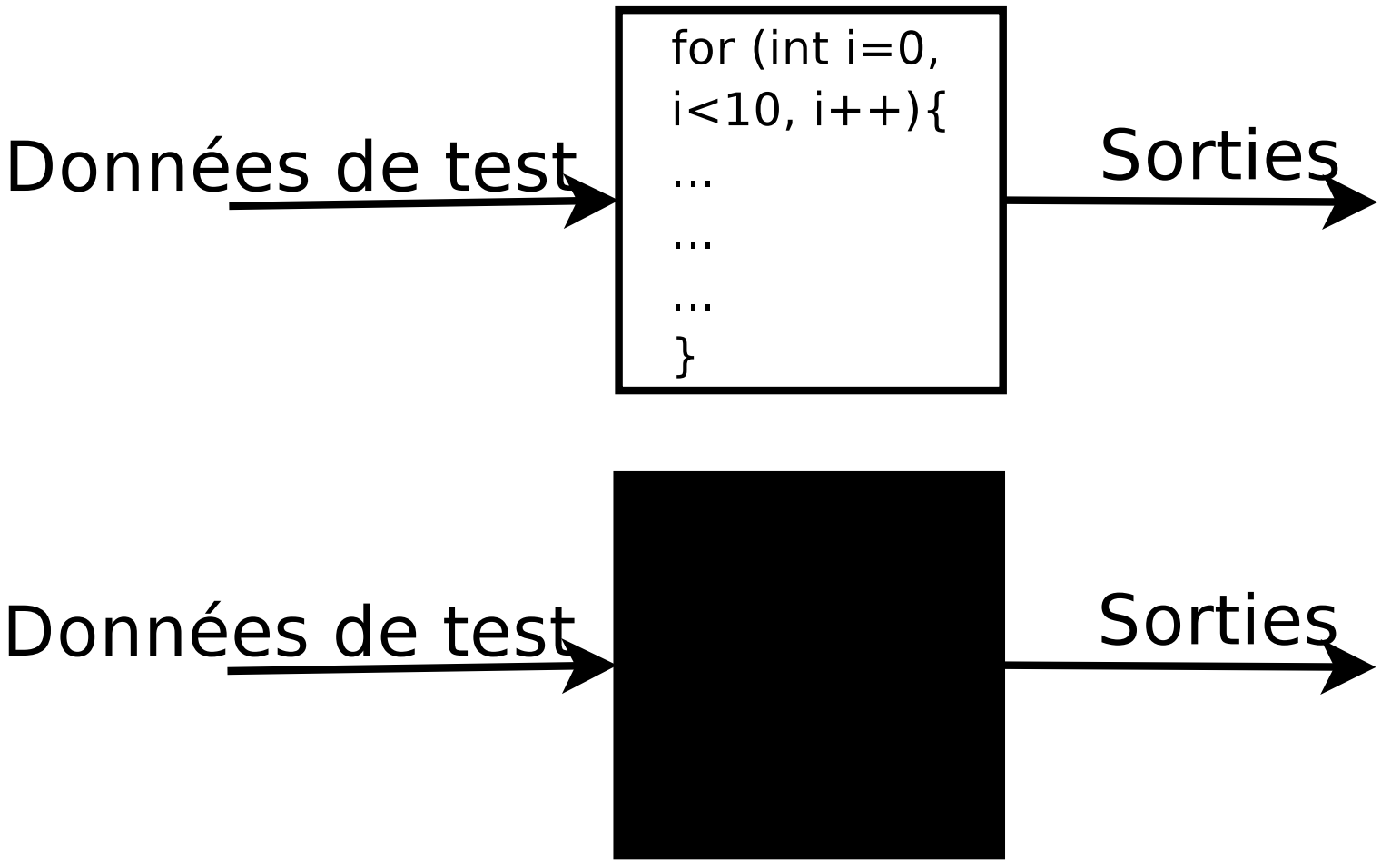
Différents niveaux d'accessibilité

- Test boîte noire (souvent fonctionnel)
- Test boîte blanche (souvent structurel)
- Test boîte grise ?

Plusieurs types classiques

- test fonctionnel
- test de non-régression
- test de montée en charge

Boîte blanche et boîte noire



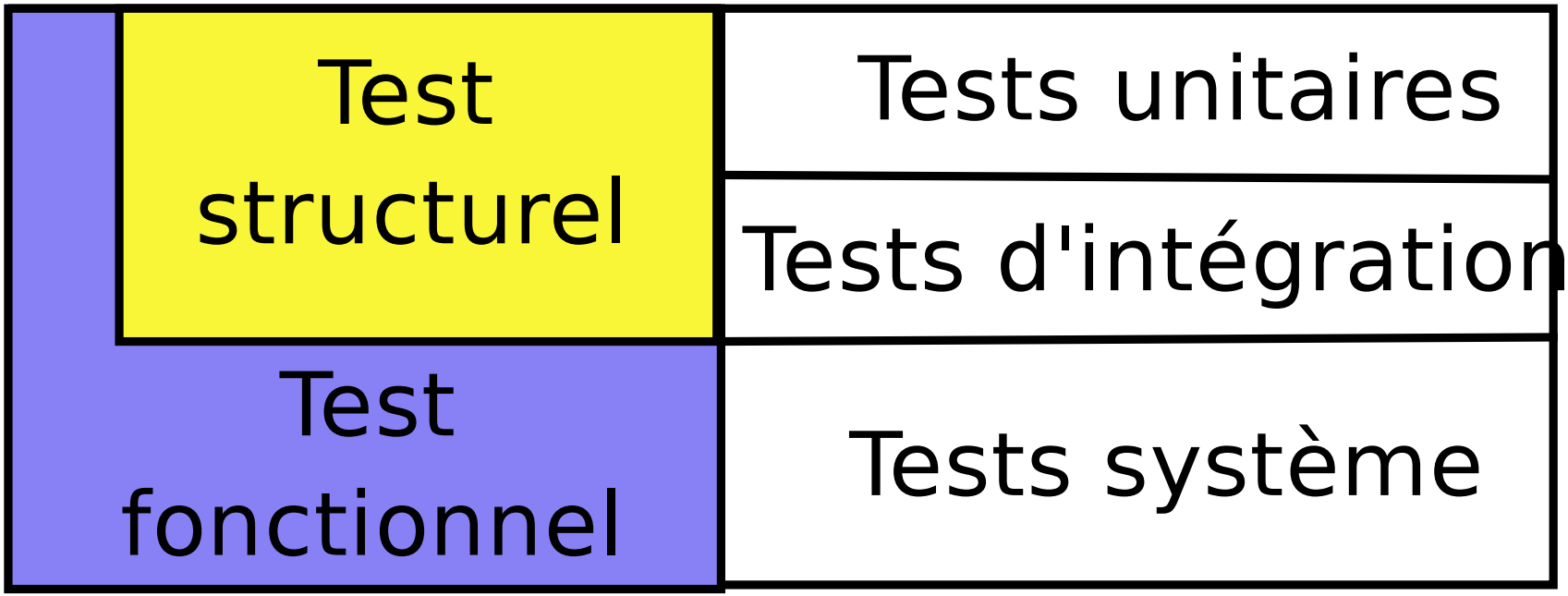
Le test fonctionnel

- Technique en général boîte noire
- On se base sur un modèle du programme issu des spécifications
  - informelles (ex. description en langage naturel)
  - semi-formelles (ex. modèles UML)
  - formelles (machines B, IOLTS, ...)

Le test structurel

- Technique boîte blanche
- On se base sur un modèle du code source du programme
  - Le modèle est une représentation de la structure
  - On utilise beaucoup la théorie des graphes pour couvrir le modèle notamment

Techniques et échelles



Complémentarité fonctionnel / structurel

```
function sum (x,y : integer) : integer;
begin
  if (x = 600) and (y = 500) then sum := x-y
  else sum := x+y;
end
```

- Une technique fonctionnelle a peu de chances de trouver l'erreur
- Une technique structurelle trouvera facilement la donnée de test (x=600, y=500)

Complémentarité fonctionnel / structurel

```
prod(int i, inj)
 int k;
 if (i==2)}{
   k:=i<<1;//décalage à gauche, multiplication par 2
 else
   faire i fois l'addition de j
return k;
```

Spécification : renvoie le produit de i par j

Complémentarité fonctionnel / structurel

```
prod(int i, inj)
 int k;
 if (i==2)}{
   k:=i<<1;//décalage à gauche, multiplication par 2
 else
   faire i fois l'addition de j
return k;
```

Spécification : renvoie le produit de i par j

Fonctionnel

On choisit (i=0,j=0)->0 et (i=10, j=100)->1000 ——> OK

Complémentarité fonctionnel / structurel

```
prod(int i, inj)
 int k;
 if (i==2)}{
   k:=i<<1;//décalage à gauche, multiplication par 2
 else
   faire i fois l'addition de j
return k;
```

Spécification : renvoie le produit de i par j

Fonctionnel

On choisit (i=0,j=0)->0 et (i=10, j=100)->1000 ——> OK

Structurel

On choisit au moins une donnée qui passe par le cas i=2 :
(i=2,j=0)->1 ——> NOK

Le test fonctionnel

- On cherche à savoir
  - si toutes les fonctionalités requises sont présentes ...
  - ... et correctes

Le test de robustesse

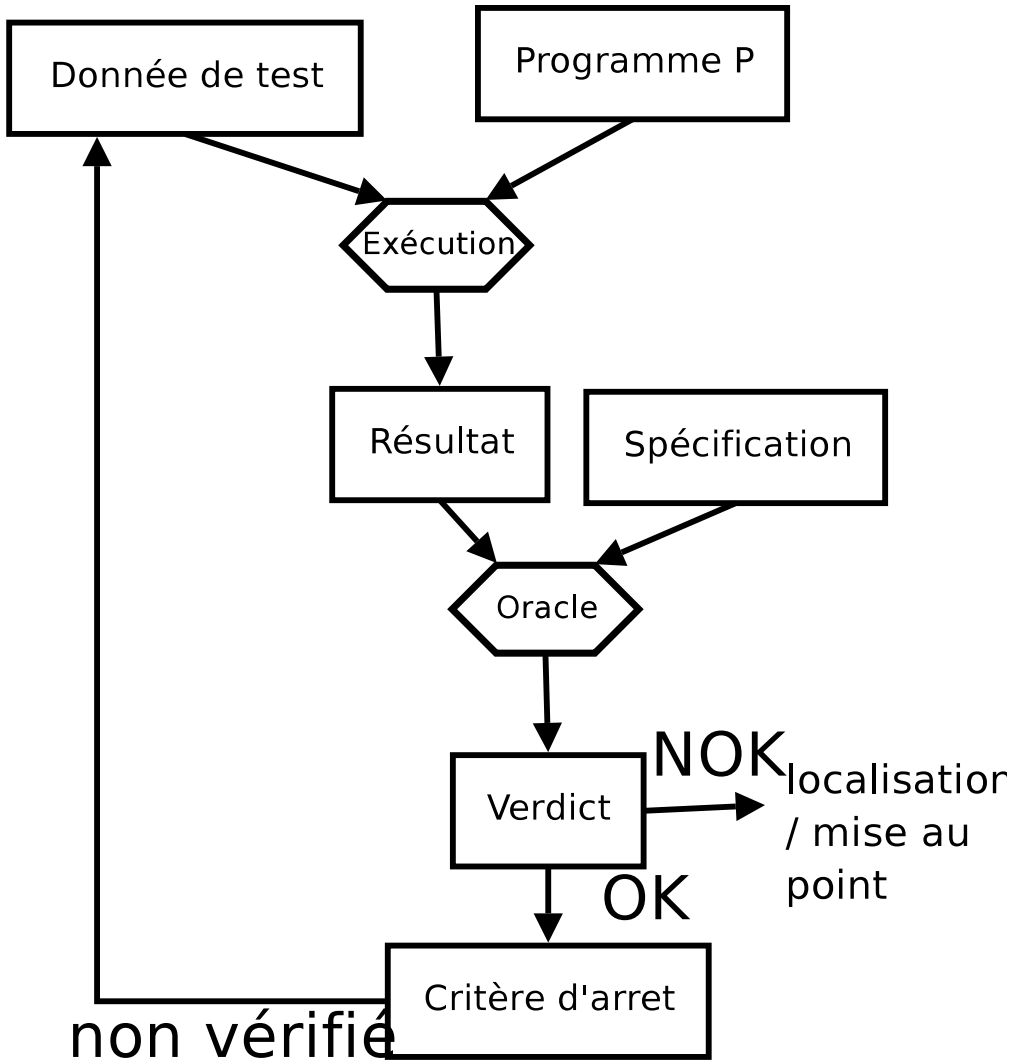
- On cherche à savoir si le système est robuste
- Par exemple,
  - on entre des entrées invalides
  - on ferme violemment le programme
  - on jette l'ordinateur par la fenêtre (test militaire)
- Et on regarde comment le programme se comporte ...

Le test de non-régression

- On cherche à savoir si on n'a pas perdu des propriétés en cours de route ...
  - après un ajout de fonctionnalité
  - après la correction d'une erreur
  - après une optimisation
- En général, on relance les tests qui passaient précédemment
  - d'où l'intérêt de les avoir stockés
  - et d'avoir automatisé l'exécution !

Sommaire

Processus


Donnée de test
Programme P
Exécution
Résultat
Spécification
Oracle
Verdict
NOK
localisation
/ mise au
point
OK
Critère d'arret
non vérifié

Vocabulaire

Oracle

- Aussi appelé fonction d'oracle
- Permet de déterminer si le test a réussi ou échoué
  - ie si le résultat obtenu est celui attendu

Critère d'arrêt

- Permet de déterminer si on a fini de tester

Les difficultés

La génération des données de test

- Comment les choisir ? Sur quels critères ?
- Si on en choisit trop, c'est long / cher !
- Il existe des techniques de génération automatique

L'oracle

- Comment savoir si ce qu'on a obtenu est correct ?
  - faire le calcul à la main ?
  - utiliser un autre programme ?
  - en théorie : utiliser la spécification ...
  - en pratique : utiliser les propriétés du programme, versions antérieure

Les difficultés

Le critère d'arrêt

- Comment savoir quand il n'est plus nécessaire de tester ?
  - on ne trouve plus d'erreurs depuis 5 minutes ?
  - on n'a plus de temps ?
  - on a passé 10h à tester ?
  - on a exécuté une fois chaque intruction ?
  - on a fait au moins 3 tours dans chacune des boucles ?

Bonnes pratiques

- Conserver les tests, ne jamais les jeter !
- Automatiser au maximum :
  - la génération de test
  - l'exécution
  - l'oracle

Les outils

- Générateurs de test
  - pas toujours avec oracle, à partir de différentes formes de spécifications ou en boîte blanche
- Les pilotes de test
  - permettent d'automatiser le lancement des tests, de créer des rapports de test
- Frameworks xUnit
- Outils de gestion de doublures de test
- Outils de mesure de couverture du code
- Outils de monitoring

Sommaire

Sommaire

Qu'est-ce qu'une doublure de test ?

- La définition d'objets factices se substituant lors du test aux objets réels
- Des objets qui remplacent des objets réels pour faciliter le test (des objets) de leur environnement
- Ces objets peuvent être écrits à la main (leur classe) ou générés.

Pourquoi des doublures de test ?

- L'environnement du SUT est complexe ou coûteux à mettre en place (environnement matériel, base de données, ...).
- Mise en place de situations exceptionnelles difficiles à déclencher (out of memory, ...).
- L'environnement du SUT n'est pas encore disponible ou fiabilisé.
- Le SUT appelle du code lent.
- Le SUT fait appel à des méthodes non déterministes (fonction de l'heure, de nombres générés aléatoirement, ...)
- on utilise alors des doublures à la place de tout ou partie de l'environnement qui pose problème.

Types de doublures

- Dummy
- Stub
- Fake
- Spy
- Mock

Pas de définition consensuelle ... Dans la suite, définitions principalement inspirées de celles de Martin Fowler.

Dummy

- Objets vides qui n'ont pas de fonctionalités implémentées
- Les dummies sont "transmis" mais jamais réellement utilisés.
- En général ils sont utilisés pour remplir des listes de paramètres.

Fake

- Le fake implémente de manière simpliste le comportement attendu d'une classe.
- Le fake est plus générique que le stub : il n'est pas spécifique à un test.
- Le fake met en place des raccourcis qui le rendent inutilisable en production

Spy

- Un spy est une doublure capable de vérifier l'utilisation qui en est faite.
- Par exemple : appel au moins une fois de telle méthode avec tel paramètre.

Bouchons de test (stubs)

- Un bouchon de test est une classe utilisée pour en simuler une autre.
- Il fournit des réponses pré-définies aux appels réalisés lors du test.
- Le bouchon de test est écrit grâce à la connaissance de la classe à simuler (boîte blanche).

Doublure et simulacre de test (mocks)

- Les mocks sont des objets pré-programmés avec des pré-suppositions qui forment la spécification des appels qu'ils sont censés recevoir.
- Le testeur configure le simulacre de manière à lui donner le comportement souhaité.
- Le code du test met en place le simulacre, le configure, puis l'utilise pour paramétrer le SUT.
- Le mock permet de réaliser une vérification comportementale. Par exemple que telle méthode a bien été appelée.
- Le mock est proche d'un stub espion.

Sommaire

Mockito

- Outil permettant de générer des doublures
- Le testeur peut facilement donner à la doublure le comportement recherché.
- Le testeur peut facilement faire une vérification comportementale après exécution (le côté espion du mock)

Principe général : création d'un mock

Quand et pourquoi ?

- Quand on veut éviter (pour des raisons vues précédemment) de faire appel au vrai environnement du SUT
- Par exemple quand certains éléments de l'environnement dépendent du temps, ou d'un alea

Principe général : création d'un mock

Comment on fait ?

- Dans le test, au lieu d'utiliser une instance d'une classe "réelle" de l'environnement (et qu'on souhaite remplacer), on demande à Mockito d'utiliser une doublure.
- On paramètre la doublure pour qu'elle ait un comportement permettant le test.
- On écrit le test, qui utilisera donc la doublure.

Principe général : création d'un mock

- Création d'un mock : utilisation de la méthode statique **mock** ou de l'annotation @mock
- Nécessite la classe à mocker ou son interface
- Mockito, crée moi une instance de mock pour cette classe ou cette interface

```
1  import static org.mockito.Mockito.*;
2  ...
3   C mock1 =mock(C.class); // C est une classe ou une interface
4   C mock2=mock(C.class , "nom");
5   @Mock C mock3;
6   @Mock(name="nom2") C mock4;
```

Remarques bassement techniques sur la création des mocks

- Pour utiliser les annotations mockito : ajouter `@RunWith(MockitoJUnitRunner.class)` sur la classe de test
- On ne peut pas utiliser l'annotation `@Mock` sur une variable locale (d'une méthode de test), seulement sur un attribut (normal, c'est une annotation)

Mais que fait un mock fraîchement créé ?

Par défaut pas grand chose ...

Ses méthodes retournent quand on les appelle :

- pour les numériques : 0
- pour les booléens : false
- pour les collections : collections vides
- pour les objets quelconques : null

D'où la nécessité de paramétrer le mock

- pour qu'il réponde des choses utiles pour le test

Principe général : paramétrage d'un mock

- On décrit le comportement du mock avec la méthode méthode `when`
- On exprime quelque chose du genre : Mockito, quand (when !) le mock recevra tel appel, alors il faut répondre ceci
- On peut faire des vérifications comportementales avec la méthode `verify`
- On vérifie quelque chose du genre : Mockito, est-ce que telle méthode a bien été appelée au moins une fois avec tel paramètre ?
- On peut spécifier des comportements à vérifier un peu complexes grâce à des `matchers`

Spécification du comportement du mock : méthodes avec retour

Cas d'une méthode avec retour ; valeur unique

```
1    interface I{ int m();} // the interface to mock
2    ...
3    @Mock I myMock; // the mock
4    ...
5    when(myMock.m()).thenReturn(42);
```

- Mockito, quand myMock recevra un appel à la méthode m, retourne 42.

Spécification du comportement du mock : méthodes avec retour

Cas d'une méthode avec retour ; valeurs successives

```
1    interface I{ int m();} / the interface to mock
2    ...
3    @Mock I myMock; // the mock
4    ...
5    when(myMock.m()).thenReturn(42, 43, 44);
```

▶ Mockito, quand myMock recevra un appel à la méthode m, retourne 42 au premier appel, puis 43 au deuxième appel, puis 44 au 3ème appel.

Spécification du comportement du mock, méthode avec paramètres

Spécifier le comportement selon la valeur reçue en paramètre

```
1    interface I{ int m(int i);} // the interface to mock
2    ...
3    @Mock I myMock; // the mock
4    ...
5    when(myMock.m(1)).thenReturn(42);
6    when(myMock.m(42)).thenReturn(1);
```

▶Mockito, quand myMock recevra l'appel à la méthode m avec pour paramètre 1, retourne 42, et avec comme paramètre 42, retourne 1.

Spéc. du comportement du mock pour lever des exceptions

Cas d'une méthode avec retour avec levée d'exception

```
1  interface I{ int m() throws E;} // the interface to mock
2  ...
3  @Mock I myMock;
4  ...
5  when(myMock.m()).thenThrow(new E());
```

▶Mockito, quand myMock reçoit un appel à m, jette une exception de type E

Cas d'une méthode sans retour avec levée d'exception

```
1  interface I{ void m(int i) throws E;} // the interface to mock
2  ...
3  @Mock I myMock; // the mock
4  ...
5  doThrow(New E()).when(myMock).m(1);
```

▶Mockito, quand myMock reçoit un appel à m avec pour paramètre 1, jette une exception de type E

Spécification du comportement du mock, exemple de combinaison

Cas d'une méthode avec paramètres ; combinaison de then

```
1  interface I{ int m(int i) throws E;}
2  ...
3  @Mock I myMock;
4  ...
5  when(myMock.m(1)).thenReturn(42).thenthrow(new E());
6  when(myMock.m(42)).thenReturn(1).thenReturn(99);
```

▶Mockito, quand myMock recevra un appel à m avec comme param 1, jette une exception de type E, et quand tu reçois un appel à m avec pour param 42, retourne 99.

Vérification du comportement : verify

Vérifier qu'une méthode est appelée 3 fois

```
1    verify(mock1, times(3)).m();
```

Vérifier qu'une méthode est appelée au moins/au plus 3 fois

```
1    verify(mock1, atLeastOnce()).m();
2       verify(mock1, atMost(3)).m();
```

Vérifier qu'une méthode n'est jamais appelée

```
1    verify(mock1, never()).m();
```

Vérification du comportement : verify

Vérifier l'ordre d'appel

```
1  InOrder ordre =inOrder(mock1, mock2);
2  ordre.verify(mock1).m();
3  ordre.verify(mock2).m();
```

Mock et spy

- On veut espionner un objet instance d'une classe réelle (pas un mock) ...
- ... et éventuellement d'en altérer le comportement
- Par exemple pour juste mocker une méthode (remplacer le comportement réel par un autre, et grader tous les autres comprtements réels)
- On utilise alors un spy, dont le comportement par défaut est celui de la classe de l'objet espionné

Spy

```
1  @Spy LinkedList<String> spy;
2  ...
3  spy.add(``ajout1'');
4  spy.add(``ajout2'');
5  verify(spy).add("ajout1");
6  when(spy.isEmpty()).thenReturn(false);
7  assertFalse(spy.isEmpty());
```

Limites et remarques

Limites

- On ne peut pas mocker les méthodes privées, ni les méthodes final et ou static avant la version 3.4 de Mockito
- On ne peut pas mocker les méthodes equals et hashcode (utilisées en interne par Mockito)
- On ne peut pas mocker les classes final (avant la version 3.4) ou anonymes

Remarques

- Un comportement non utilisé du mock ne provoque pas d'erreur
- Verify : si la vérification échoue, le test échoue.

Argument matchers

- Permettent une spécification de paramètres flexible dans les when ou les verify
- Ne plus utiliser `Matchers` (deprecated pour cause de conflit de nommage avec hamcrest)
- `https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/ArgumentMatchers.html`

```
1  public interface I{public int m(int i);}
2  @Mock I mock;
3  ...
4  when(mock.m(anyInt())).thenReturn(42);
5  assertEquals(mock.m(12), 42);
```

- autres argument matchers : eq, contains, ...
- limitation : si on utilise des argument matchers, tous les arguments doivent être des argument matchers.

Sommaire

PowerMock

- Framework de test permettant de contourner les limitations classiques des frameworks de mock comme Mockito.
- Avec PowerMock, on peut :
  - mocker des méthodes privées, statiques, et/ou final
  - mocker des classes final
  - ...
- “Please note that PowerMock is mainly intended for people with expert knowledge in unit testing. Putting it in the hands of junior developers may cause more harm than good.”

Préparation de la classe de test et de la classe mockée

```
1  package tpmocks;
2  import org.junit.Assert;
3  import org.junit.Test;
4  import org.junit.runner.RunWith;
5  import org.mockito.Mockito;
6  import org.powermock.api.mockito.PowerMockito;
7  import org.powermock.core.classloader.annotations.PrepareForTest;
8  import org.powermock.modules.junit4.PowerMockRunner;
9
10 @RunWith(PowerMockRunner.class)
11 @PrepareForTest({A.class})
12 public class MesTestsPowerMockSurA {...}
```

Mock d'une méthode finale

```
1    public class A{ final String jeSuisFinal(){return "AF";} ...}
2    ...
3  @Test
4  public void mockFinalTest() {
5      A mock = PowerMockito.mock(A.class);
6      final String resultMock = "valeur mockee";
7      Mockito.when(mock.jeSuisFinal()).thenReturn(resultMock);
8      // on verifie que la methode final a bien ete mockee
9      Assert.assertEquals(mock.jeSuisFinal(), resultMock);
```

Mock d'une méthode statique

```
1    public class A{ static String jeSuisStatique(){return "AS";} ...}
2    ...
3  @Test
4  public void mockStaticTest() {
5      PowerMockito.mockStatic(A.class);
6      String resultMock = "valeur mockee";
7      Mockito.when(A.jeSuisStatique()).thenReturn(resultMock);
8      // on verifie que la methode statique a bien ete mockee
9      Assert.assertEquals(A.jeSuisStatique(), resultMock);
10     }
```