MGL7460 — Automne 2016

Tests unitaires, cadres de tests et TDD

Guy Tremblay
Professeur

Département d'informatique
UQAM
http://www.labunix.uqam.ca/~tremblay

22 septembre 2016

Contenu

1. Introduction/motivation

Rappel : Parmi les premières choses à faire, quand on développe du code de façon professionnelle…

«*[Y]ou need to get the development infrastructure environment in order. That means adopting (or improving) the fundamental Starter Kit practices :*

- ?
- ?
- ?

? *needs to come before anything else. It's the first bit of infrastructure we set up on any project.*»

«*Practices of an Agile Developer—Working in the Real World*», Subramaniam & Hunt, 2006.

Rappel : Parmi les premières choses à faire, quand on développe du code de façon professionnelle…

«*[Y]ou need to get the development infrastructure environment in order. That means adopting (or improving) the fundamental Starter Kit practices :*

- *Version control*
- *Unit Testing*
- *Build automation*

Version control needs to come before anything else. It's the first bit of infrastructure we set up on any project.»

«*Practices of an Agile Developer—Working in the Real World*», Subramaniam & Hunt, 2006.

Rappel : Quelques *tips* de Hunt & Thomas

48. Design to Test
    Start thinking about testing before you write a line of code.

49. Test Your Software, or Your Users Will
    Test ruthlessly. Don't make your users find bugs for you.

62. Test Early. Test Often. Test Automatically.
    Tests that run with every build are much more effective than test plans that sit on a shelf.

63. Coding Ain't Done 'Til All the Tests Run
    'Nuff said.

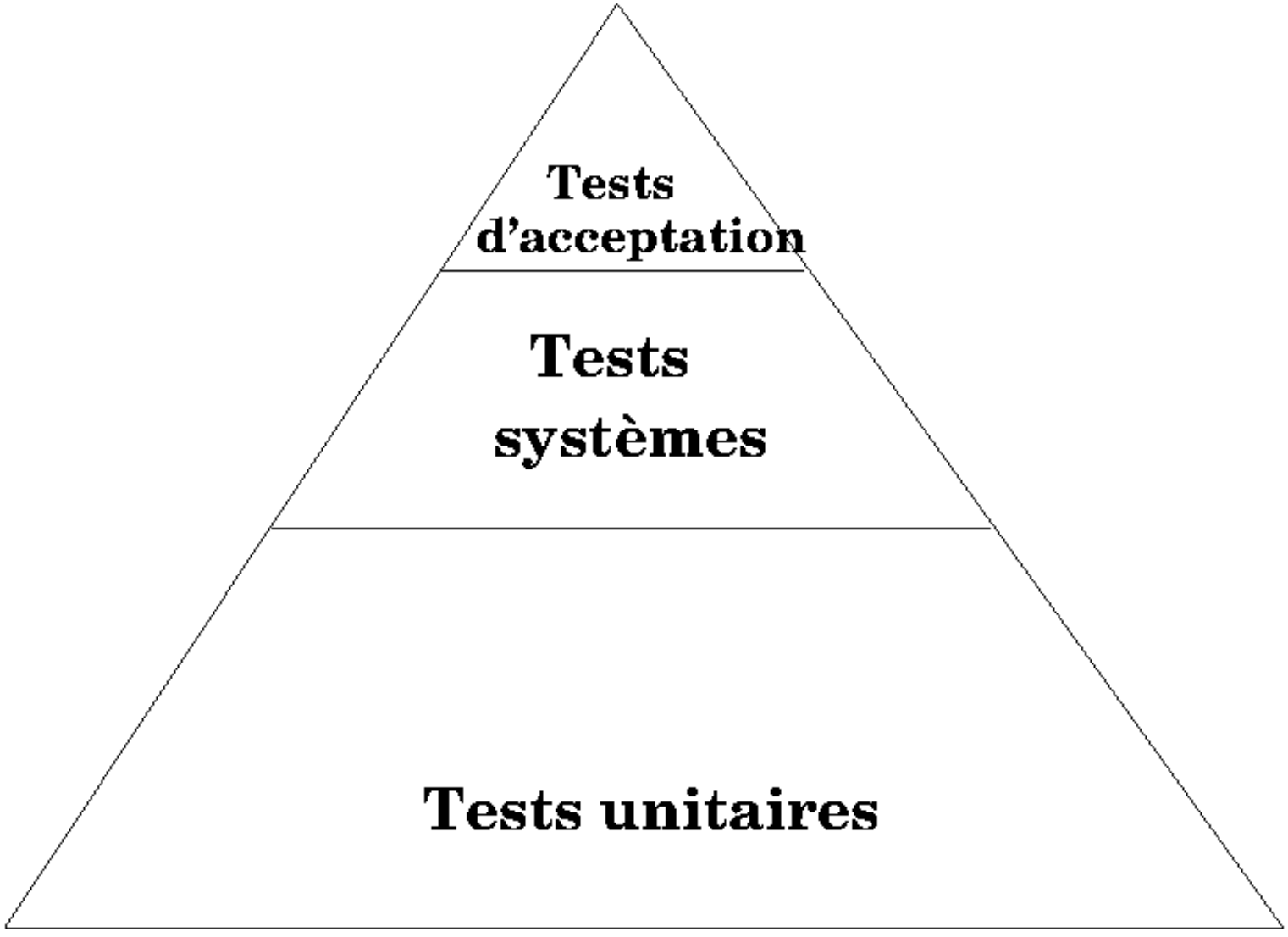
66. Find Bugs Once
    Once a human tester finds a bug, it should be the last time a human tester finds that bug. Automatic tests should check it from then on.

2. Les tests unitaires

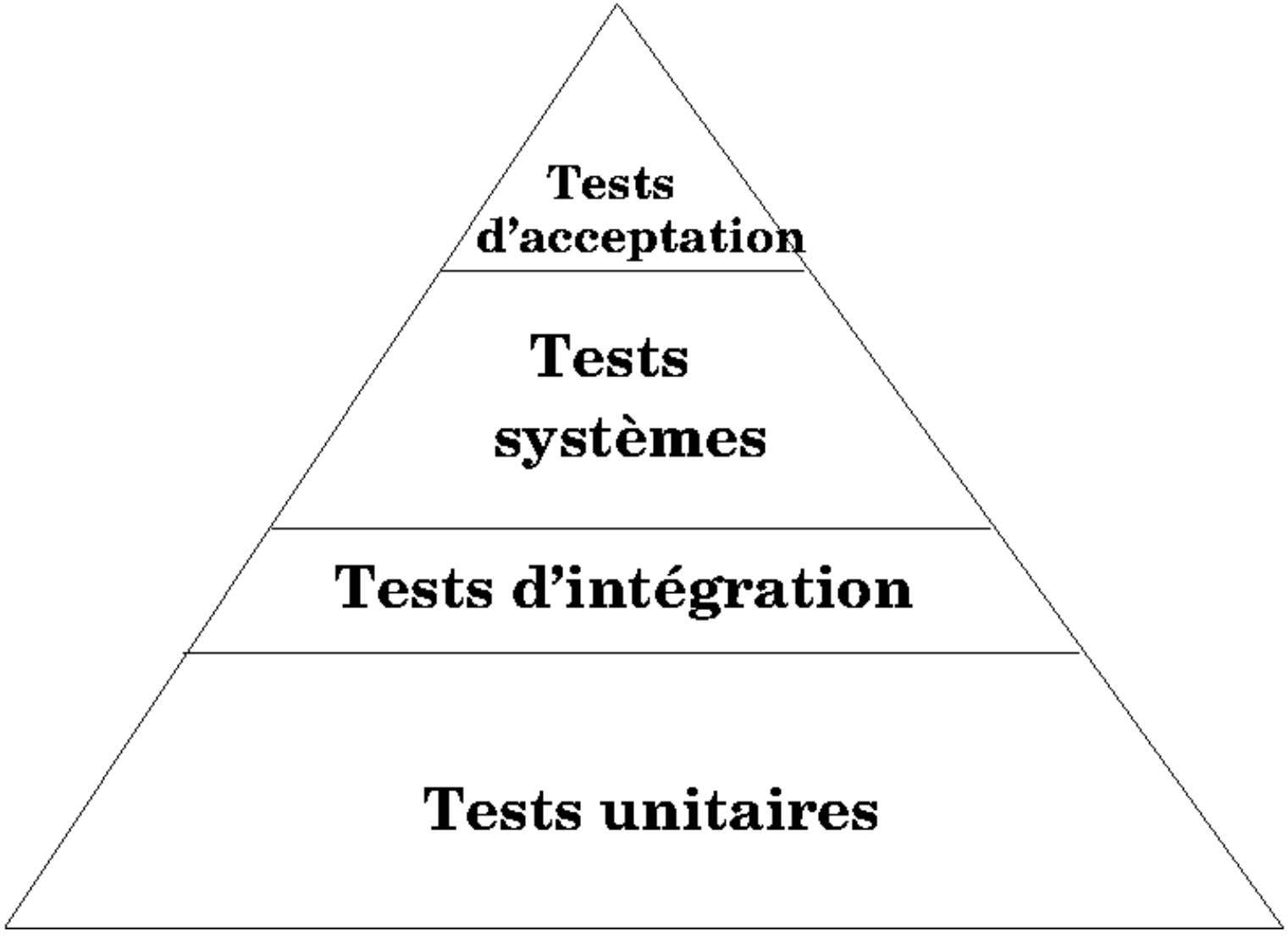
Pourquoi les tests sont-ils importants ?

Les différents niveaux de tests

Les différents niveaux de tests

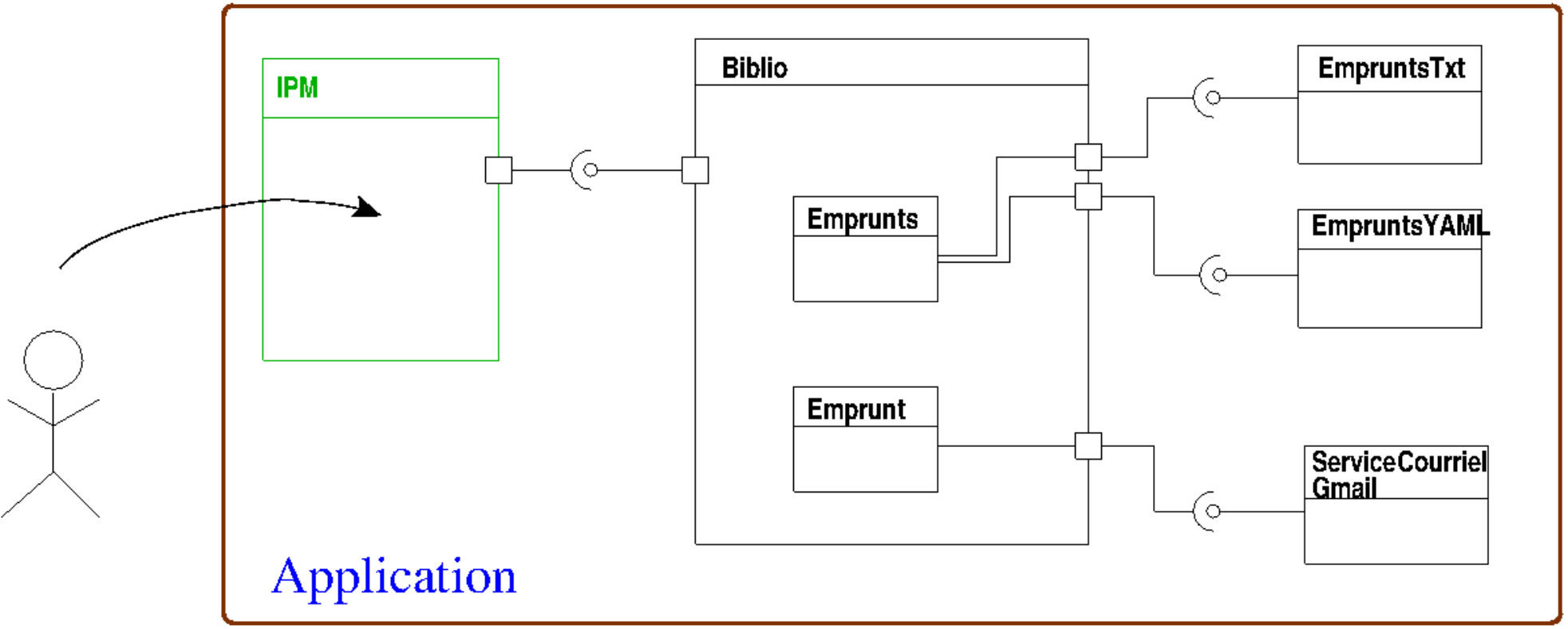


Les différents niveaux de tests



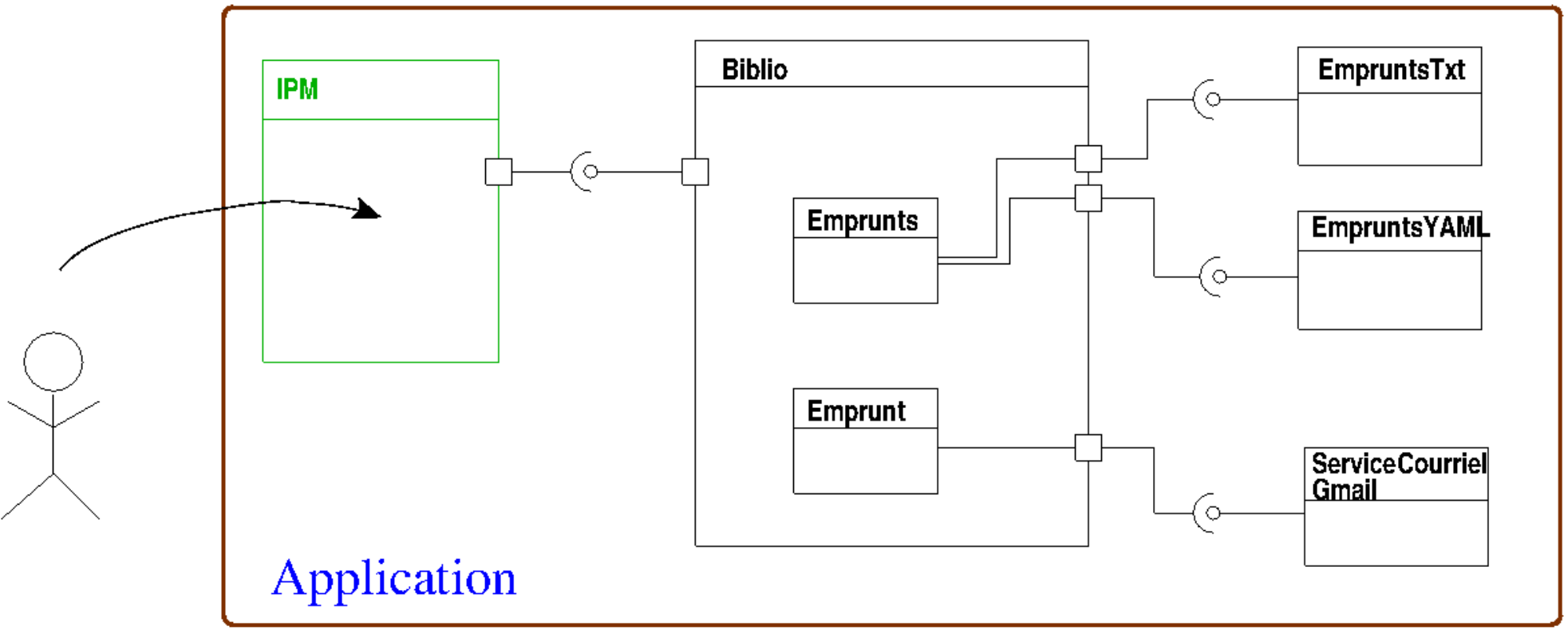
Les différents niveaux de tests

Soit une application composée de plusieurs modules et composants



Les différents niveaux de tests

Soit une application composée de plusieurs modules et composants



Question : Qu'entend-on par tests de type «boite blanche» vs. «boite noire» (*white box vs. black box*) ?

Deux grands types de tests

Test de type «boite blanche»

- Fondés sur la structure interne du composant

⇒ Conçus en connaissant le code du composant

⇒ Écrits par les développeurs

- Tests unitaires et d'intégration

Tests de type «boite noire»

- Fondés sur l'interface (publique) du composant

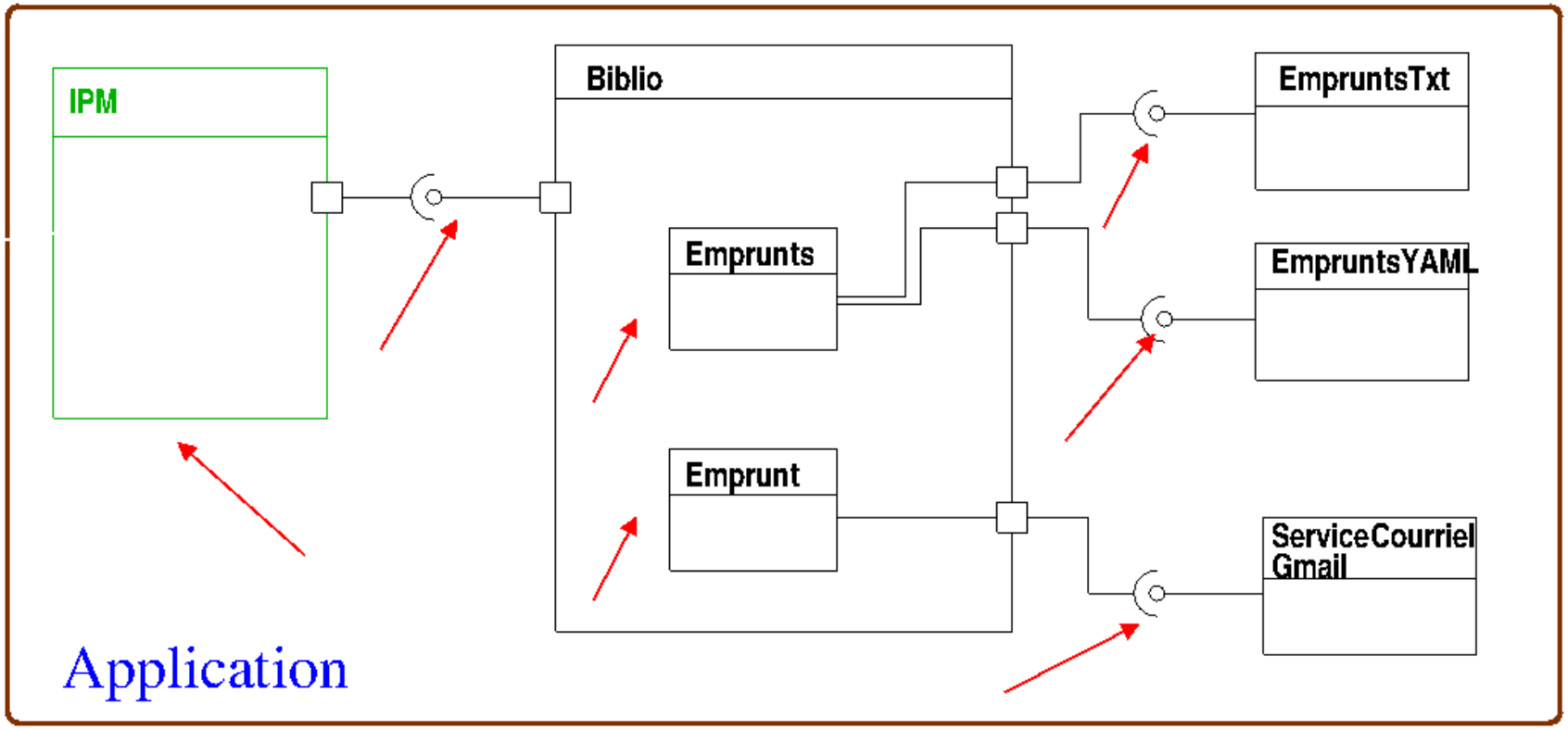
⇒ Conçus sans connaître le code du composant

⇒ Écrits tôt et indépendamment des développeurs

- Typique : Tests de système et d'acceptation

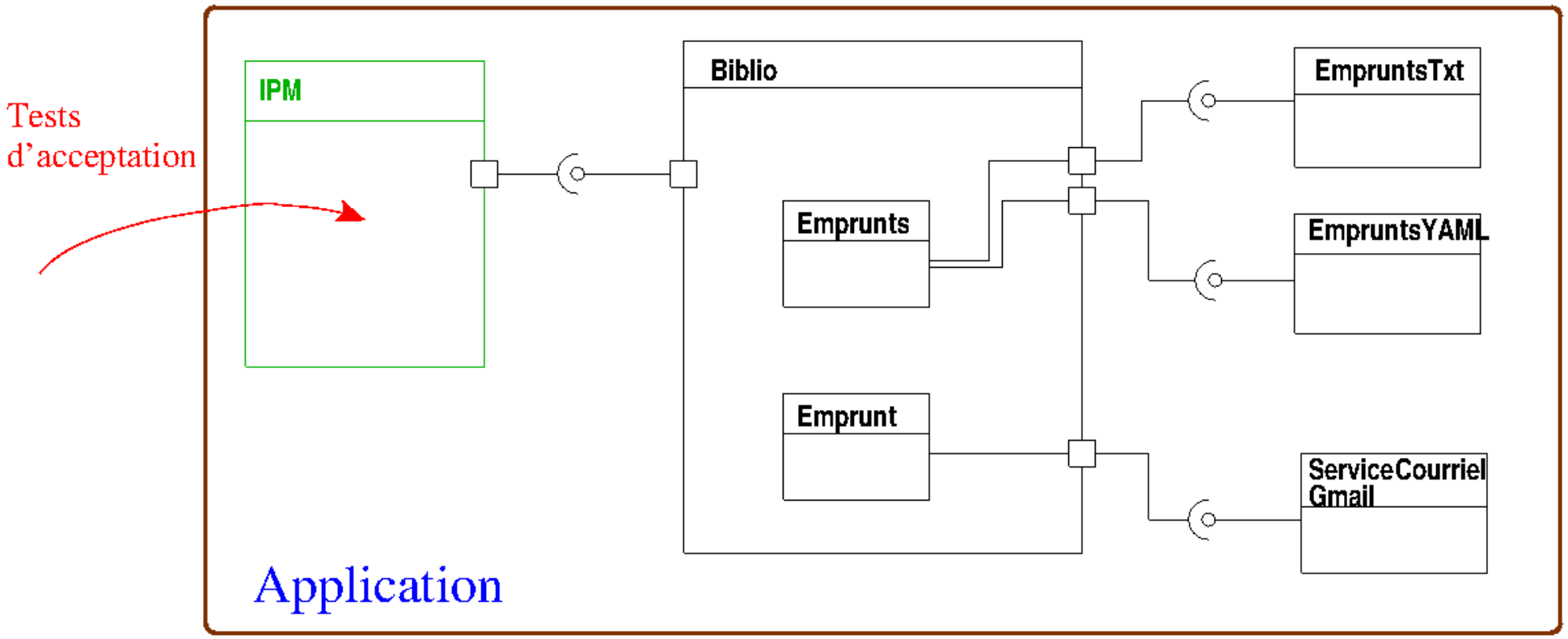
Les différents niveaux de tests

Les tests unitaires pour les modules et composants de cette application doivent tester chaque morceau, donc **de l'intérieur** du logiciel — tests «boite blanche»



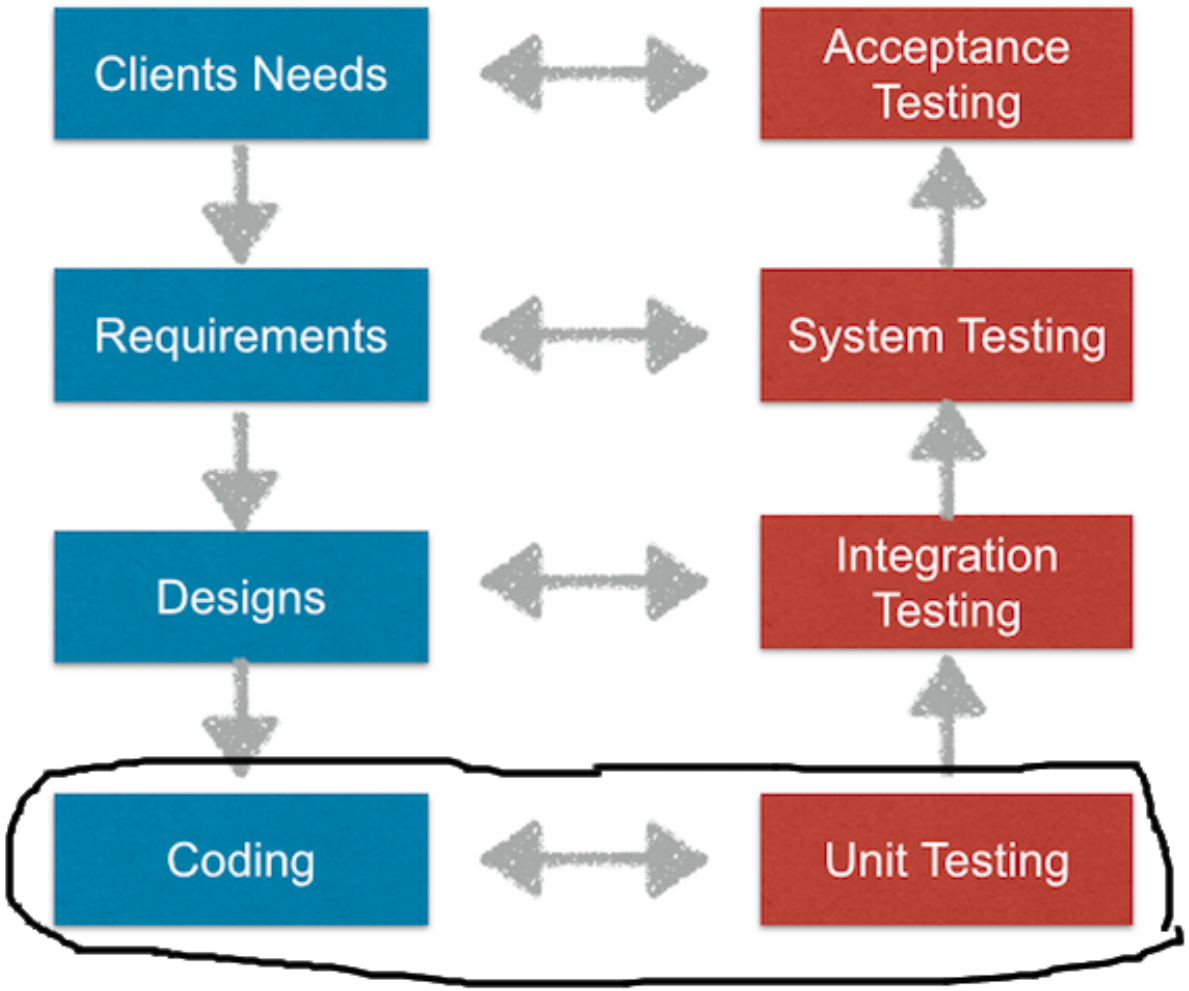
Les différents niveaux de tests

Les tests d'acceptation pour cette application doivent tester le logiciel dans son ensemble, **de l'extérieur** — tests «boite noire»



Note : Idem pour les tests de système !

Les différents niveaux de tests



Source: `http://usersnap.com/blog/types-user-acceptance-tests-frameworks/`

Les différents niveaux de tests

Tests unitaires

Vérification du fonctionnement d'un composant (procédure, fonction, méthode, classe, module) de façon indépendante des autres composants.

Tests de système

Vérification du fonctionnement du système dans son ensemble.

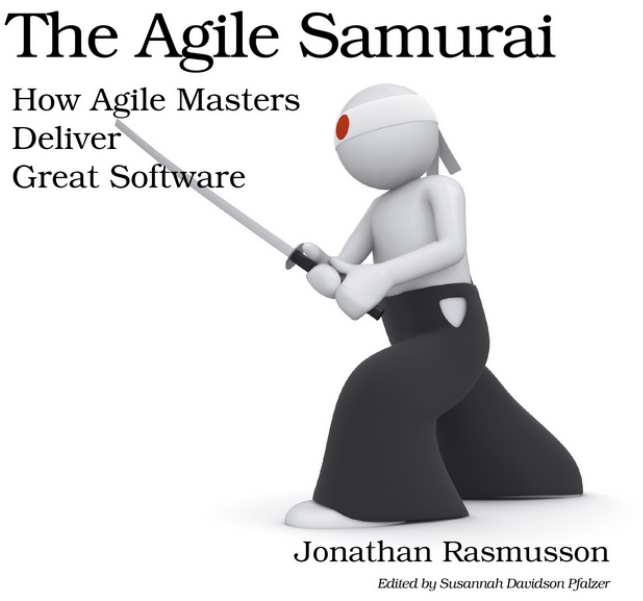
Tests d'acceptation

Vérification, par le «client», du fonctionnement du système dans son ensemble — tests fonctionnels = *user facing*.

Les tests unitaires

Une définition des tests unitaires

Unit tests are small, method-level tests developers write every time they make a change to the software to prove the changes they made work as expected.



Pourquoi les tests unitaires sont importants

- Clarifient le contrat que doit respecter le composant
  - Préconditions qui régissent l'utilisation d'une méthode
  - Postconditions assurées par l'exécution d'une méthode
  - Exceptions pouvant être signalées
- Assurent le bon fonctionnement du composant avant son intégration avec d'autres composants
- Fournissent des exemples d'utilisation du composant

Pourquoi les tests unitaires sont importants (suite)

- Permettent d'obtenir rapidement du feedback

$\Rightarrow$ Réduisent le temps consacré (perdu ☹) à déboguer

- Assurent la non-régression

Pourquoi les tests unitaires sont importants (suite)

- Permettent d'obtenir rapidement du feedback

⇒ Réduisent le temps consacré (perdu ☹) à déboguer

- Assurent la non-régression

Pourquoi les tests unitaires sont importants (suite)

- Permettent d'obtenir rapidement du feedback

$\Rightarrow$ Réduisent le temps consacré (perdu ☹) à déboguer

- Assurent la non-régression

- Permettent de faire du refactoring avec confiance

Pourquoi les tests unitaires sont importants (suite)

- Permettent d'obtenir rapidement du feedback

$\Rightarrow$ Réduisent le temps consacré (perdu ☹) à déboguer

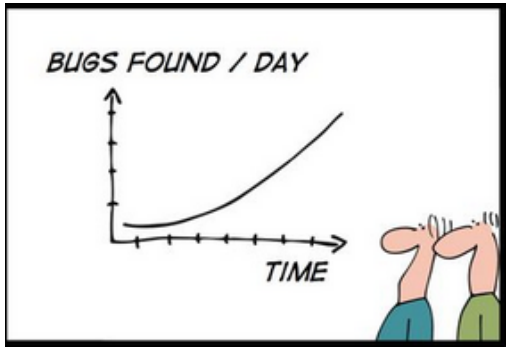
- Assurent la non-régression

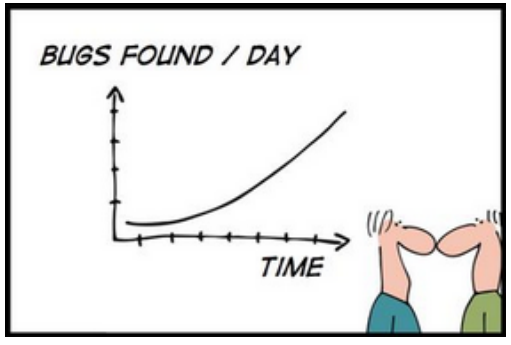
- Permettent de faire du refactoring avec confiance

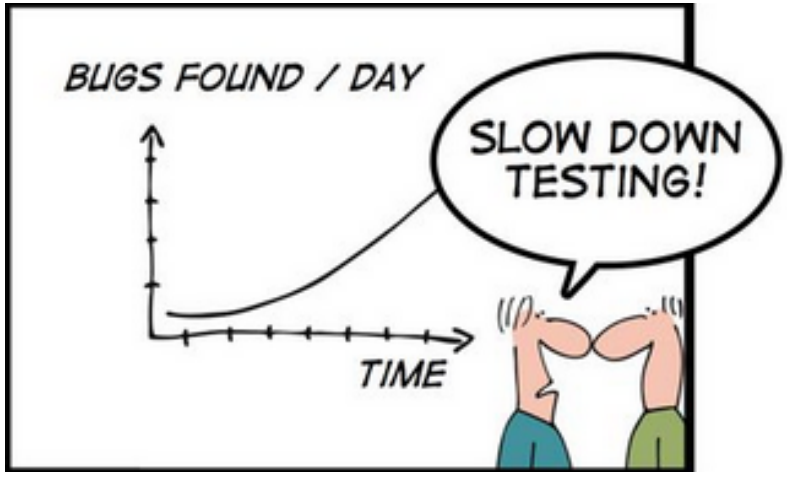
Tests de (non-)régression : tests d'un programme préalablement testé, après une modification, pour s'assurer que des défauts n'ont pas été introduits ou découverts dans des parties non modifiées du logiciel.

`https://fr.wikipedia.org/wiki/Test_de_r%C3%A9gression`

Les tests développés tôt sont cruciaux pour trouver rapidement les bogues







Source: `http://geek-and-poke.com`

Les tests unitaires font partie intégrante du «code»

```
$ ls -l Biblio
total 248
[...] 16 déc  2014 biblio.gemspec
[...] 10 nov  2014 biblio.rdoc
[...] 16 déc  2014 bin/
[...] 10 nov  2014 config/
[...] 10 nov  2014 features/
[...] 10 nov  2014 Gemfile
[...] 16 déc  2014 Gemfile.lock
[...] 10 nov  2014 lib/
[...]  8 jun 15:50 Rakefile
[...] 10 nov  2014 README.rdoc
[...] 10 nov  2014 test/
```

Les tests unitaires peuvent nécessiter autant ou même plus de LOC que le code lui-même

Oto, un logiciel d'aide à la correction de programmes

Tests unitaires :

Fichiers `*.rb`	**Nb. fichiers**	**KLOC**
Tous fichiers `*.rb`	247	19
Fichiers de tests	114	8

Tests fonctionnels :

Fichiers	**KLOC**
651	25

Note : KLOC = Mille lignes de code (*lines of code*)

Les tests unitaires peuvent nécessiter autant ou même plus de LOC que le code lui-même

SQLite

- Version 3.8.10
- Bibliothèque écrite en C
- SLOC = *Source Lines Of Code*

$\Rightarrow$ exclut lignes blanches et commentaires

Partie du Code SQLite	**KSLOC**
Code fonct.	94
Tests	?

Source : «*How SQLite Is Tested*»

`https://www.sqlite.org/testing.html`

Les tests unitaires peuvent nécessiter autant ou même plus de LOC que le code lui-même

SQLite

- Version 3.8.10
- Bibliothèque écrite en C
- SLOC = *Source Lines Of Code*

$\Rightarrow$ exclut lignes blanches et commentaires

Partie du Code SQLite	**KSLOC**
Code fonct.	94
Tests	91515

Source : «*How SQLite Is Tested*»

`https://www.sqlite.org/testing.html`

Les tests unitaires doivent être exécutés souvent

Pour identifier les problèmes rapidement

Le plus tôt on trouve un problème ou une erreur (un *bogue*), le plus facile il est de le localiser et le corriger
$\Rightarrow$ "*Code a little, test a little*".

Pour assurer la non régression

Si on effectue des modifications, il faut s'assurer «que rien n'a été brisé», que ce qui fonctionnait avant fonctionne encore.
$\Rightarrow$ *tests de régression* — ou "*tests de non régression*".

Old tests never die, they just become regression tests

Les tests unitaires doivent être exécutés souvent et automatiquement

Pour identifier les problèmes rapidement

Le plus tôt on trouve un problème ou une erreur (un *bogue*), le plus facile il est de le localiser et le corriger
$\Rightarrow$ "*Code a little, test a little*".

Pour assurer la non régression

Si on effectue des modifications, il faut s'assurer «que rien n'a été brisé», que ce qui fonctionnait avant fonctionne encore.
$\Rightarrow$ *tests de régression* — ou "*tests de non régression*".

Old tests never die, they just become regression tests

Souvent $\Rightarrow$
Facilement et automatiquement

Les cadres de tests

Différentes stratégies d'exécution des tests

Exécution manuelle

Pour un logiciel très, très simple, où les tests n'auront pas besoin d'être exécutés souvent — est-ce ça existe vraiment ?

Exécution avec scripts de tests

⇒ Programmes dédiés à l'exécution des tests.

- Utilisation de `make` et de fichiers `makefile`.
- Utilisation de *shell scripts*.

Exécution avec un cadre de tests

Cadre de tests = Outil pour définir des programmes de tests puis les exécuter de façon automatique.

Les cadres de tests sont des outils qui permettent l'exécution automatique des tests

Caractéristiques générales

- Facilitent l'écriture des tests — et leur association au composant testé.
- Permettent d'exécuter de façon automatique une ou plusieurs suites de tests.
- Reposent sur l'utilisation d'assertions $\Rightarrow$ *Self-checking tests* :
  - Si tout est ok, alors «peu de bruit»
  - S'il y a des erreurs, alors plus d'informations et détails

- Permettent d'organiser les tests de façon structurée (hiérarchique)
- Fournissent des mécanismes pour la construction d'échafaudage, d'objets complexes, *stubs*, *mocks*, etc.

Les cadres de tests permettent d'avoir des *Self-Checking Tests*

A Self-Checking Test is one that has encoded within it everything that it needs to verify that the expected outcome is correct. Self-checking tests apply the "Hollywood Principle" ("don't call us, we will call you") to running tests. That is, the `Test Runner` *only "calls us" when a test did not pass thus making a clean test run have zero manual effort .*

Source: `http://xunitpatterns.com/Goals%20of%20Test%20Automation.html#Self-Checking%20Test`

L’ancêtre de tous les cadres de tests = ?

L'ancêtre de tous les cadres de tests = sUnit

«*Simple Smalltalk Testing With Patterns*», K. Beck, 1989 :

> *I recommend that developers write their own unit tests, one per class. The framework supports the writing of suites of tests, which can be attached to a class. I recommend that all classes respond to the message "testSuite", returning a suite containing the unit tests. I recommend that developers spend 25–50% of their time developing tests.*

Source : `http://live.exept.de/doc/online/english/tools/misc/testfram.htm`

Qui est Kent Beck ?

Source : `http://www.azquotes.com/author/31849-Kent_Beck`

Qui est Kent Beck ?

Source : `http://www.azquotes.com/author/31849-Kent_Beck`

- XP : « *Extreme Programming Explained—Embrace Change* »
- TDD : « *Test-Driven Development—By Example*
- JUnit

L'ancêtre de tous les cadres de tests = sUnit

Exemple `sUnit` (1989 !)

```
SetTestCase >> setUp
    empty := Set new.
    full := Set with: #abc with: 5

SetTestCase >> testAdd
    empty add: 5.
    self should: [empty includes: 5]

SetTestCase >> testRemove
    full remove: 5.
    self should: [full includes: #abc].
    self shouldnt: [full includes: 5]

SetTestCase >> testIllegal
    self should: [self errorSignal
                       handle: [:ex | true]
                       do: [empty at: 5. false]]
```

Source : `http://live.exept.de/doc/online/english/tools/misc/testfram.htm`

De (très !) nombreux cadres de tests sont disponibles

L'outil le plus connu = JUnit

- Cadre de tests pour Java
- Développé et popularisé par les promoteurs de XP (*eXtreme Programming*) — notamment, Kent Beck.

De (très !) nombreux cadres de tests sont disponibles

L'outil le plus connu = JUnit

- Cadre de tests pour Java
- Développé et popularisé par les promoteurs de XP (*eXtreme Programming*) — notamment, Kent Beck.
- A rendu «populaire» l'écriture de tests unitaires et l'utilisation de cadres de test

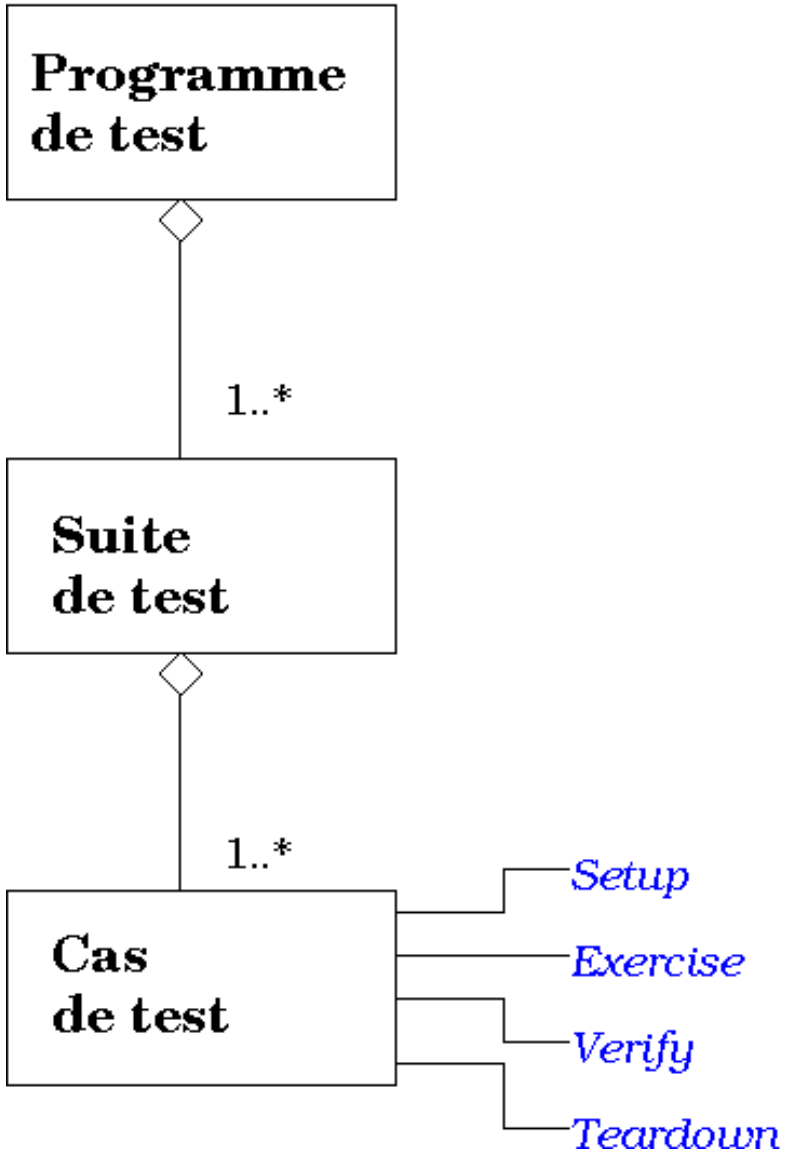
De (très !) nombreux cadres de tests sont disponibles

Des cadres équivalents existent pour d'autres langages

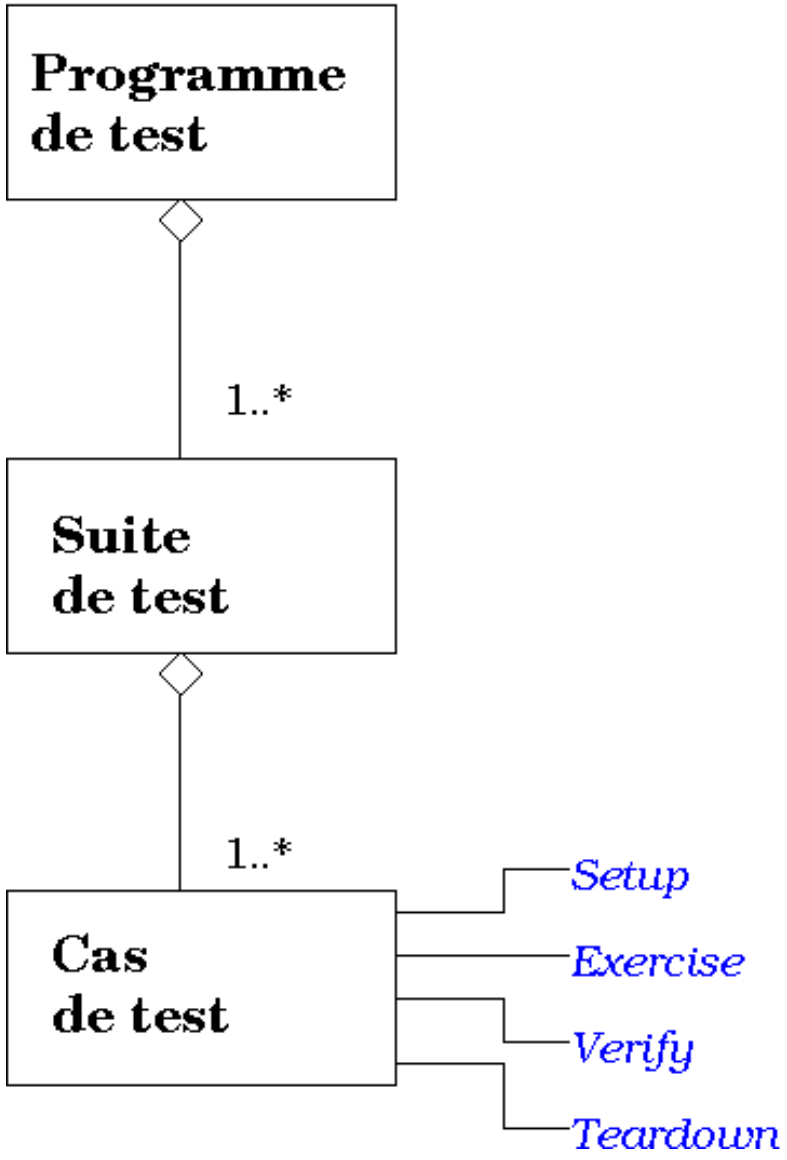
- `xUnit` = famille de cadres de tests avec des variantes pour divers langages : Ada, C, C++, Eiffel, Java, Perl, Python, etc..
- Le site suivant en répertorie... plus de 150 : `http://c2.com/cgi/wiki?TestingFramework`
- Par ex., cadres de tests pour le langage C : `GUNit`, `Check`, `MinUnit`, `CUnit`, `CuTest` + 9 autres.

Les programmes de test

Organisation typique d'un programme de test défini dans un cadre de tests



Organisation typique d'un programme de test défini dans un cadre de tests



- (*Setup*) On crée des objets
- (*Exercise*) On appelle la méthode à tester
- (*Verify*) On vérifie que le résultat — ou l'effet produit — est celui désiré
- (*Teardown*) On nettoie

Organisation typique d'un programme de test défini dans un cadre de tests : *Four-Phase Test* ⋆

Extrait de «*xUnit Test Patterns*» de G. Meszaros

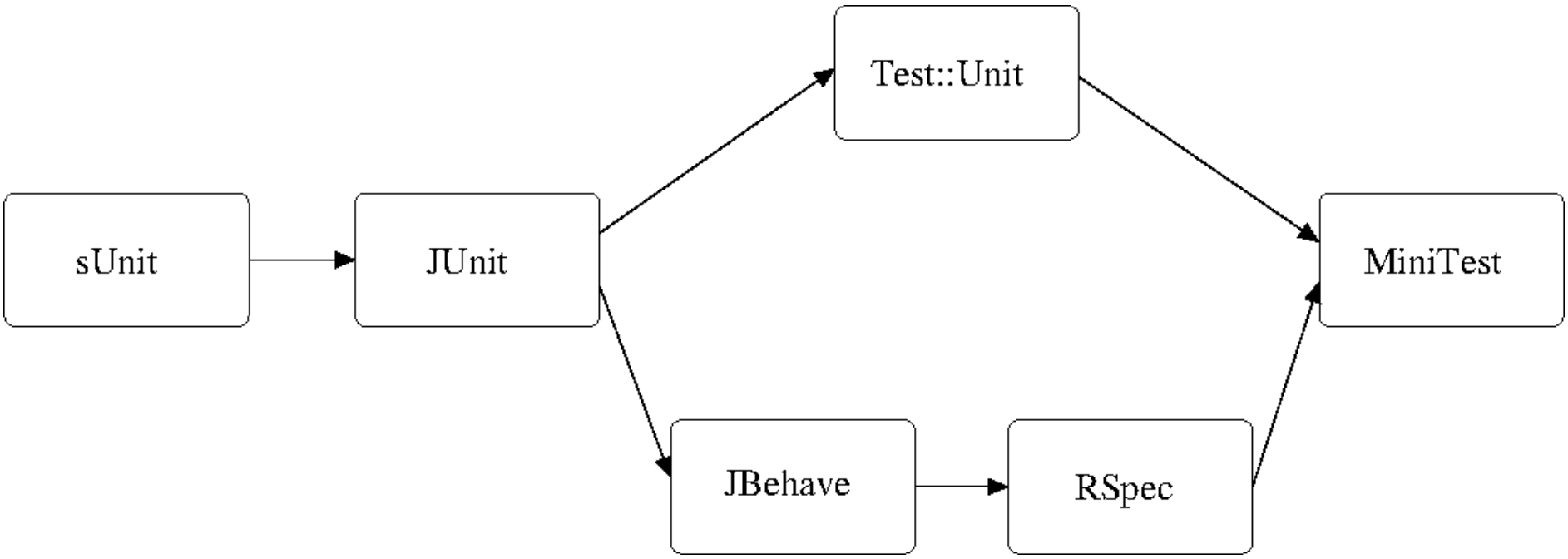
[Question :] How do we structure our test logic to make what we are testing obvious ?
Answer : **We structure each test with four distinct parts executed in sequence.**

1. *set up the test fixture (the "before" picture) that is required for the SUT to exhibit the expected behavior as well as anything you need to put in place to be able to observe the actual outcome [. . .].*
2. *interact with the SUT.*
3. *do whatever is necessary to determine whether the expected outcome has been obtained.*
4. *tear down the text fixture to put the world back into the state in which we found it.*

Note : SUT = System Under Test

3. `MiniTest`, un cadre de tests pour Ruby

Les ancêtres de `MiniTest`



Les deux approches possibles en `MiniTest` pour les cas de test

Style JUnit : Avec «*assertions*»

```
def test_foo_blah_blah_blah
  ...
  assert_equal expected, actual
end
```

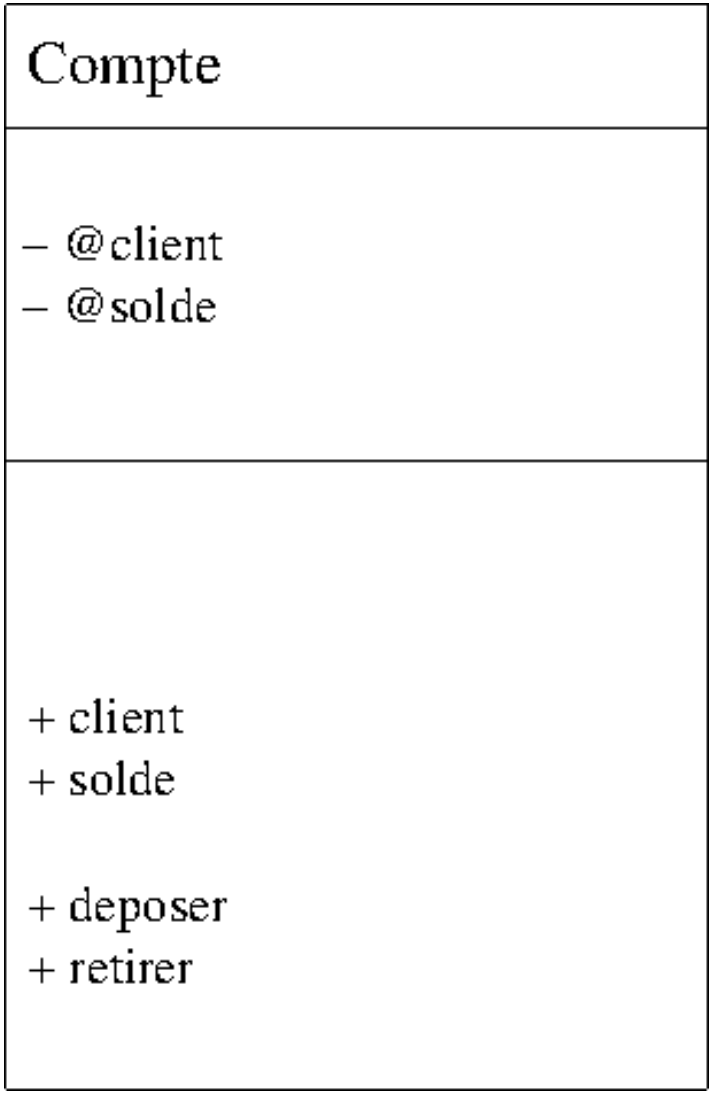
Style RSpec : Avec «*expectations*»

```
describe "#foo"
  it "blah blah blah" do
    ...
    actual.must_equal expected
  end
end
```

Exemple :
Classe à tester = `Compte` bancaire simple

```ruby
class Compte
  attr_reader :solde, :client

  def initialize( client, solde_initial )
    @client, @solde = client, solde_initial
  end

  def deposer( montant )
    @solde += montant
  end

  def retirer( montant )
    fail "Solde insuffisant" if montant > solde

    @solde -= montant
  end
end
```

Un diagramme de classe UML pour la classe `Compte`



3.1 Approche avec assertions (style JUnit/Test : :Unit)

Exemple :
Classe de test avec *assertions* (1/2)

```
class TestCompte < Minitest::Test
  def setup
    @solde_initial = 100
    @c = Compte.new( "Guy T.", @solde_initial )
  end

  def test_solde_retourne_le_solde_initial
    assert_equal @solde_initial, @c.solde
  end

  def test_deposer_ajoute_le_montant_indique_au_solde_du_compte
    # Exercise.
    solde_initial = @c.solde
    @c.deposer( 100 )

    # Verify.
    assert_equal (solde_initial + 100), @c.solde
  end
  ...
```

Exemple :
Classe de test avec *assertions* (1/2)

```
class TestCompte < Minitest::Test
  def setup
    @solde_initial = 100
    @c = Compte.new( "Guy T.", @solde_initial )
  end

  def test_solde_retourne_le_solde_initial
    assert_equal @solde_initial, @c.solde
  end

  def test_deposer_ajoute_le_montant_indique_au_solde_du_compte
    # Exercise.
    solde_initial = @c.solde
    @c.deposer( 100 )

    # Verify.
    assert_equal (solde_initial + 100), @c.solde
  end
  ...
```

Exemple :
Classe de test avec *assertions* (2/2)

```
  ...
  def test_retirer_deduit_le_montant_desire_lorsque_ne_depasse
    solde_initial = @c.solde
    @c.retirer( 50 )

    assert_equal (solde_initial - 50), @c.solde
  end

  def test_retirer_vide_le_compte_lorsque_le_montant_desire_es
    @c.retirer( @c.solde )

    assert_equal 0, @c.solde
  end

  def test_retirer_signale_une_erreur_lorsque_le_montant_desir
    assert_raises(RuntimeError) { @c.retirer( 2050 ) }
  end
end
```

Exemple :
Résultat d'exécution sans erreur

```
$ ruby compte_minitest_assertions.rb
ruby compte_minitest_assertions.rb
Run options: --seed 65217

# Running:

.....

Finished in 0.001107s, 4515.3169 runs/s, 4515.3169 asser

5 runs, 5 assertions, 0 failures, 0 errors, 0 skips
```

Exemple :
Résultat d'exécution avec erreur

```
$ ruby compte_minitest_assertions.rb
Run options: --seed 19158

# Running:

F....

Finished in 0.001119s, 4468.7704 runs/s, 4468.7704 asser

  1) Failure:
TestCompte#test_deposer_ajoute_le_montant_indique_au_sol
[compte_minitest_assertions.rb:19]:
Expected: 200
  Actual: 0

5 runs, 5 assertions, 1 failures, 0 errors, 0 skips
```

Les différentes méthodes de `MiniTest` pour des *assertions* — positives vs. négatives

- #assert
- #assert_empty
- #assert_equal
- #assert_in_delta
- #assert_in_epsilon
- #assert_includes
- #assert_instance_of
- #assert_kind_of
- #assert_match
- #assert_nil
- #assert_operator
- #assert_output
- #assert_predicate
- #assert_raises
- #assert_respond_to
- #assert_same
- #assert_send
- #assert_silent
- #assert_throws

- #refute
- #refute_empty
- #refute_equal
- #refute_in_delta
- #refute_in_epsilon
- #refute_includes
- #refute_instance_of
- #refute_kind_of
- #refute_match
- #refute_nil
- #refute_operator
- #refute_predicate
- #refute_respond_to
- #refute_same

3.2 Approche avec *expectations* (style JBehave/RSpec)

À l'origine, une suggestion de D. North (2003)

Le nom d'une méthode de test devrait être une phrase qui nous éclaire sur le comportement de la méthode testée

À l'origine, une suggestion de D. North (2003)

Le nom d'une méthode de test devrait être une phrase qui nous éclaire sur le comportement de la méthode testée

Exemple :

- *NomDeLaMéthode_ÉtatTesté_RésultatOuEffetAttendu*

Le nom d'une méthode de test devrait être une phrase qui décrit le comportement attendu ⋆

Test Naming Conventions de «xUnit Test Patterns» (G. Meszaros)

Naming of Testcase Classes *and* Test Methods *is crucial for making our tests easy to find and understand. We can make the test coverage more obvious by naming each* Test Method *systematically based on what test condition is being verified. Regardless of which test method organization scheme we use, we would like the combination of the names of the test package, the* Testcase Class *and* Test Method *to convey at least the following information :*

- *The name of the SUT class.*
- *The name of the method or feature being exercise.*
- *The important characteristics of any input values related to the exercising of the SUT.*
- *Anything relevant about the state of the SUT or its dependencies.*

Source: `http://xunitpatterns.com/Organization.html#Test%20Naming%20Conventions`

Note : SUT = *System Under Test*

Le nom d'une méthode de test devrait être une phrase qui décrit le comportement attendu : Contre-Exemple

JUnit 3.0

```
public void testRetirer() {
  c.retirer( c.solde() );

  assertEquals( 0, c.solde() );
}
```

Le nom d'une méthode de test devrait être une phrase qui décrit le comportement attendu : Contre-Exemple

JUnit 4.0

```
@Test
public void retirer() {
  c.retirer( c.solde() );

  assertEquals( 0, c.solde() );
}
```

Le nom d'une méthode de test devrait être une phrase qui décrit le comportement attendu : Exemple

JUnit 4.0

```
@Test
public void Retirer_LeSoldeComplet_RetourneSoldeNul() {
  c.retirer( c.solde() );

  assertEquals( 0, c.solde() );
}
```

Le nom d'une méthode de test devrait être une phrase qui décrit le comportement attendu : Exemple

Ruby (`Test::Unit`)

```ruby
def test_retirer_le_solde_complet_retourne_solde_nul
  @c.retirer( @c.solde )

  assert_equal 0, @c.solde
end
```

Le nom d'une méthode de test devrait être une phrase qui décrit le comportement attendu : Exemple

Ruby (style `RSpec`)

```
test "retirer le solde complet retourne solde nul" do
  @c.retirer( @c.solde )

  assert_equal 0, @c.solde
end
```

Suggestion de D. North (2003)

Débuter le nom d'une méthode de test par `should` aide à ce que le test soit mieux ciblé

Suggestion de D. North (2003)

Débuter le nom d'une méthode de test par `should` aide à ce que le test soit mieux ciblé

Exemple :

Retirer

- devrait retourner un solde nul lorsqu'on retire tout
- devrait échouer lorsqu'on retire plus que le solde courant
- ...

JBehave (North, 2003)

- Extension de JUnit «*which removed any reference to testing and replaced it with a vocabulary built around verifying behaviour*»
- L'étape de vérification s'exprime avec des should plutôt qu'avec des «assertions»

JBehave (North, 2003)

- Extension de JUnit «*which removed any reference to testing and replaced it with a vocabulary built around verifying behaviour*»

- L'étape de vérification s'exprime avec des should plutôt qu'avec des «assertions»

```
assert_equal resultat_attendu, resultat_obtenu
```

```
resultat_obtenu.should == resultat_attendu
```

Notation Ruby/RSpec

Adaptation de l'approche `RSpec` en `MiniTest`

`RSpec` :

```
resultat_obtenu.should == resultat_attendu
```

$\Rightarrow$

`MiniTest` :

```
resultat_obtenu.must_equal resultat_attendu
```

Exemple :
Classe de test avec *expectations* (1/2)

```
describe Compte do
  let(:solde_initial) { 100 }
  before {  @c = Compte.new( "Guy T.", solde_initial ) }

  describe ".new" do
    it "cree un compte avec le solde initial indique" do
      @c.solde.must_equal solde_initial
    end
  end

  describe "#deposer" do
    it "ajoute le montant indique au solde" do
      solde_initial = @c.solde
      @c.deposer( 100 )

      @c.solde.must_equal (solde_initial + 100)
    end
  end
```

Exemple :
Classe de test avec *expectations* (1/2)

```
describe Compte do
  let(:solde_initial) { 100 }
  before {  @c = Compte.new( "Guy T.", solde_initial ) }

  describe ".new" do
    it "cree un compte avec le solde initial indique" do
      @c.solde.must_equal solde_initial
    end
  end

  describe "#deposer" do
    it "ajoute le montant indique au solde" do
      solde_initial = @c.solde
      @c.deposer( 100 )

      @c.solde.must_equal (solde_initial + 100)
    end
  end
```

Exemple :
Classe de test avec *expectations* (1/2)

```
describe Compte do
  let(:solde_initial) { 100 }
  before { @c = Compte.new( "Guy T.", solde_initial ) }

  describe ".new" do
    it "cree un compte avec le solde initial indique" do
      @c.solde.must_equal solde_initial
    end
  end

  describe "#deposer" do
    it "ajoute le montant indique au solde" do
      solde_initial = @c.solde
      @c.deposer( 100 )

      @c.solde.must_equal (solde_initial + 100)
    end
  end
```

Exemple :
Classe de test avec *expectations* (2/2)

```
  describe "#retirer" do
    it "deduit le montant desire lorsqu'il ne depasse pas le
      solde_initial = @c.solde
      @c.retirer( 50 )

      @c.solde.must_equal (solde_initial - 50)
    end

    it "vide le compte lorsque le montant desire est egal au
      @c.retirer( @c.solde )

      @c.solde.must_equal 0
    end

    it "signale une erreur lorsque le montant desire depasse
      lambda{ @c.retirer( 2050 ) }.must_raise RuntimeError
    end
  end
end
```

Exemple :
Résultat d'exécution ordinaire et sans erreur

```
$ ruby compte_minitest_expectations.rb
Run options: --seed 22518

# Running:

.....

Finished in 0.002205s, 3628.5918 runs/s, 4989.3138 assertions/

5 runs, 5 assertions, 0 failures, 0 errors, 0 skips
```

Exemple :
Résultat d'exécution verbose et sans erreur

```
$ ruby compte_minitest_expectations.rb --verbose
Run options: --verbose --seed 479

# Running:

Compte::#retirer#test_0001_deduit le montant desire lorsqu'il
Compte::#retirer#test_0003_signale une erreur lorsque le monta
Compte::#retirer#test_0002_vide le compte lorsque le montant d
Compte::#deposer#test_0001_ajoute le montant indique au solde
Compte::.new#test_0001_cree un compte avec le solde initial in

Finished in 0.001948s, 3080.4822 runs/s, 2567.0685 assertions/

5 runs, 5 assertions, 0 failures, 0 errors, 0 skips
```

Exemple :
Résultat d'exécution verbose et sans erreur

```
$ ruby compte_minitest_expectations.rb --verbose
Run options: --verbose --seed 479

# Running:

Compte::#retirer#test_0001_deduit le montant desire lorsqu'il
Compte::#retirer#test_0003_signale une erreur lorsque le monta
Compte::#retirer#test_0002_vide le compte lorsque le montant d
Compte::#deposer#test_0001_ajoute le montant indique au solde
Compte::.new#test_0001_cree un compte avec le solde initial in

Finished in 0.001948s, 3080.4822 runs/s, 2567.0685 assertions/

5 runs, 5 assertions, 0 failures, 0 errors, 0 skips
```

Exemple :
Résultat d'exécution verbose et sans erreur

```
$ ruby compte_minitest_expectations.rb --verbose
Run options: --verbose  --seed 479

# Running:

Compte::#retirer#test_0001_deduit le montant desire lorsqu'il
Compte::#retirer#test_0003_signale une erreur lorsque le monta
Compte::#retirer#test_0002_vide le compte lorsque le montant d
Compte::#deposer#test_0001_ajoute le montant indique au solde
Compte::.new#test_0001_cree un compte avec le solde initial in

Finished in 0.001948s, 3080.4822 runs/s, 2567.0685 assertions/

5 runs, 5 assertions, 0 failures, 0 errors, 0 skips
```

Ordre d'exécution des cas de tests

Fait : L'ordre d'exécution des tests n'est pas le même que l'ordre des définitions.

C'est le comportement de plusieurs cadres de tests... dans le but d'assurer l'indépendance des tests.

Rôle du `seed` en MiniTest = définir le germe pour la génération pseudo-aléatoire de l'ordre d'exécution des tests.

Si on veut que l'ordre d'exécution respecte l'ordre de définition, alors on utilise la commande suivante :

i_suck_and_my_tests_are_order_dependent!

Les différentes méthodes de `MiniTest` pour des *expectations* — positives vs. négatives

- #must_be
- #must_be_close_to
- #must_be_empty
- #must_be_instance_of
- #must_be_kind_of
- #must_be_nil
- #must_be_same_as
- #must_be_silent
- #must_be_within_delta
- #must_be_within_epsilon
- #must_equal
- #must_include
- #must_match
- #must_output
- #must_raise
- #must_respond_to
- #must_send
- #must_throw

- #wont_be
- #wont_be_close_to
- #wont_be_empty
- #wont_be_instance_of
- #wont_be_kind_of
- #wont_be_nil
- #wont_be_same_as
- #wont_be_within_delta
- #wont_be_within_epsilon
- #wont_equal
- #wont_include
- #wont_match
- #wont_respond_to

4. Le principe d’inversion des dépendances

Propriété des tests unitaires = Indépendants les uns des autres, donc exécutés en «isolation» !

Les tests unitaires sont supposés être faits de façon «indépendante» pour chaque module, classe, composant

- Permet de mieux cerner les problèmes : on teste un morceau à la fois

- Les tests unitaires devraient s'exécuter rapidement

  $\Rightarrow$

  Un test unitaire ne devrait pas faire (trop) d'accès à des opérations externes — entrées/sorties, fichiers/BDs, services externes (e.g., courriel), etc.

Note : Les tests sont souvent exécutés, comme en `MiniTest`, dans un ordre (pseudo-)aléatoire.

Michael Feathers' «*Unit Test Rulz*» (sic)

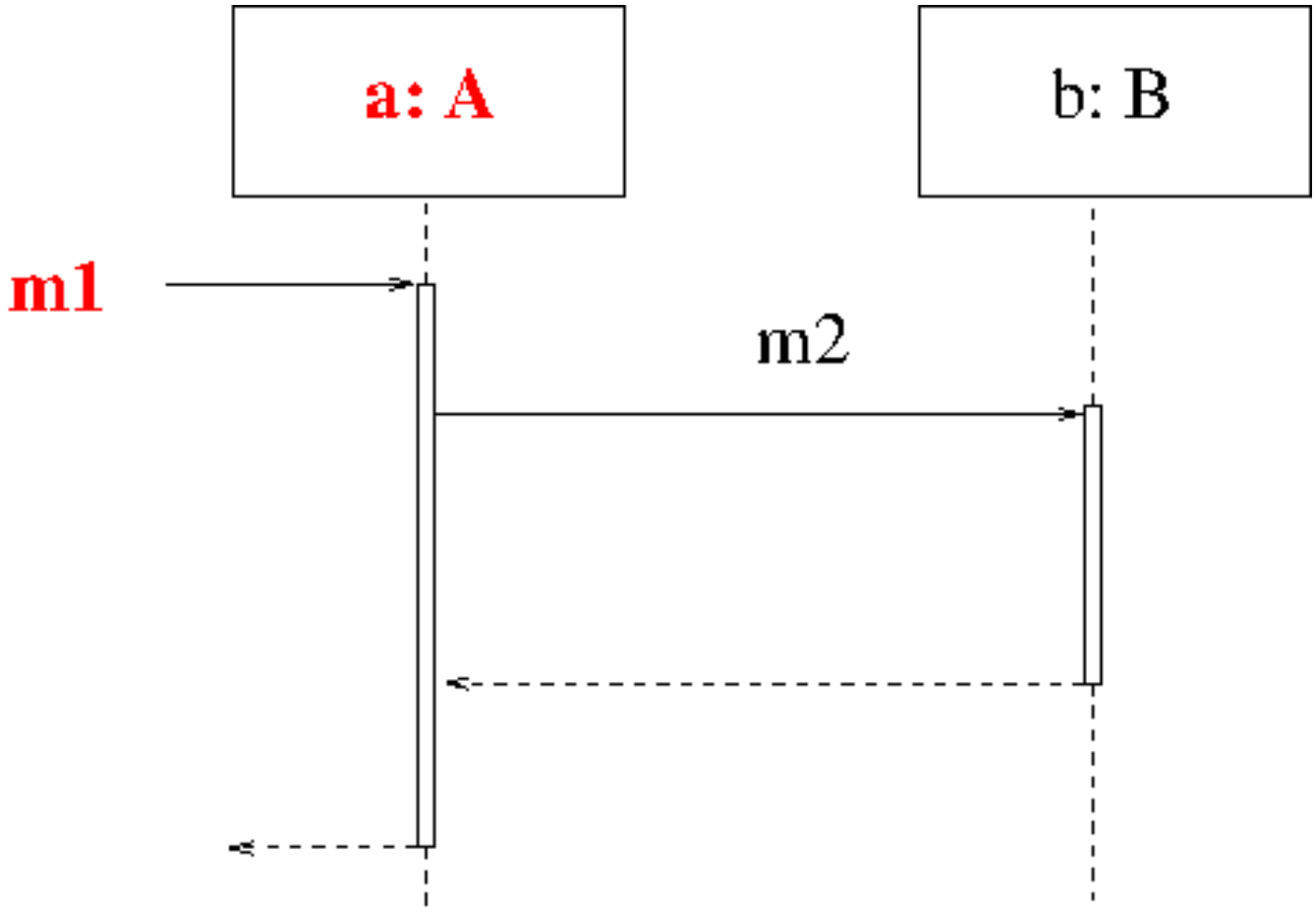
I've used these rules with a large number of teams. They encourage good design and rapid feedback and they seem to help teams avoid a lot of trouble.
A test is not a unit test if :

1. *It talks to the database*
2. *It communicates across the network*
3. *It touches the file system*
4. *It can't run correctly at the same time as any of your other unit tests*
5. *You have to do special things to your environment (such as editing config files) to run it.*

Tests that do these things aren't bad. Often they are worth writing, and they can be written in a unit test harness. However, it is important to be able to separate them from true unit tests so that we can keep a set of tests that we can run fast whenever we make our changes.

Source: `http://xunitpatterns.com/Unit%20Test%20Rulz.html`

Question : Comment fait-on pour tester un objet qui dépend d'un autre objet ?



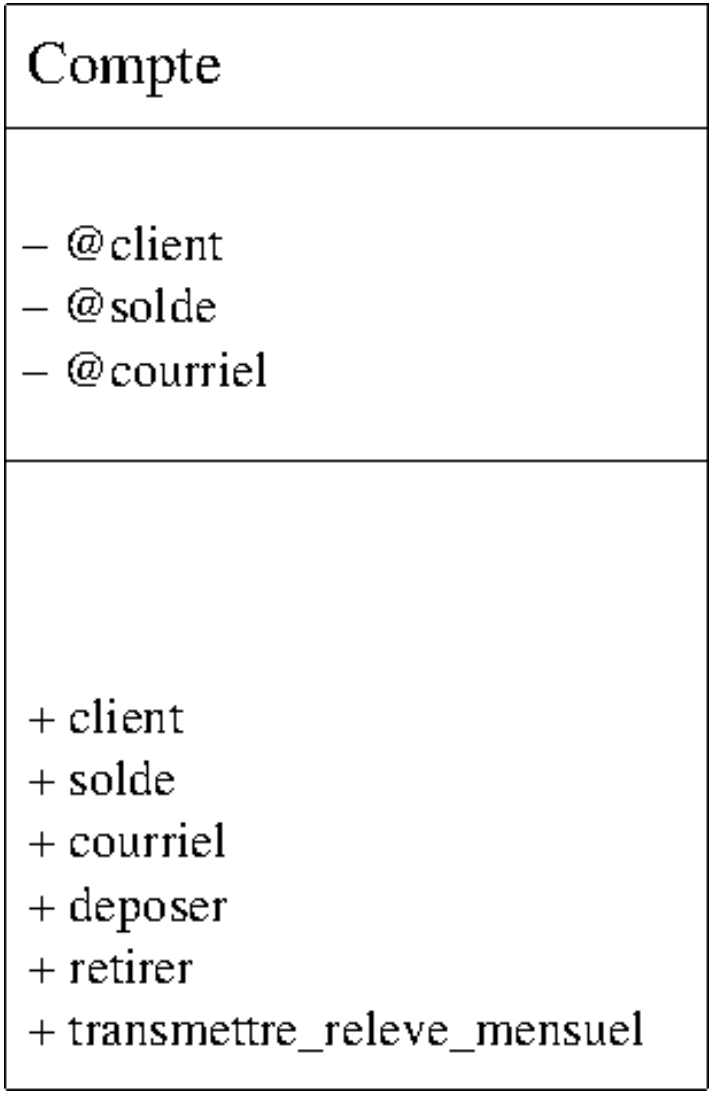
On veut tester la méthode `m1` de `A`, qui utilise `m2` de la `B` ! ?

Exemple : Envoi de relevés mensuels par courriel

Une nouvelle définition de la classe `Compte`

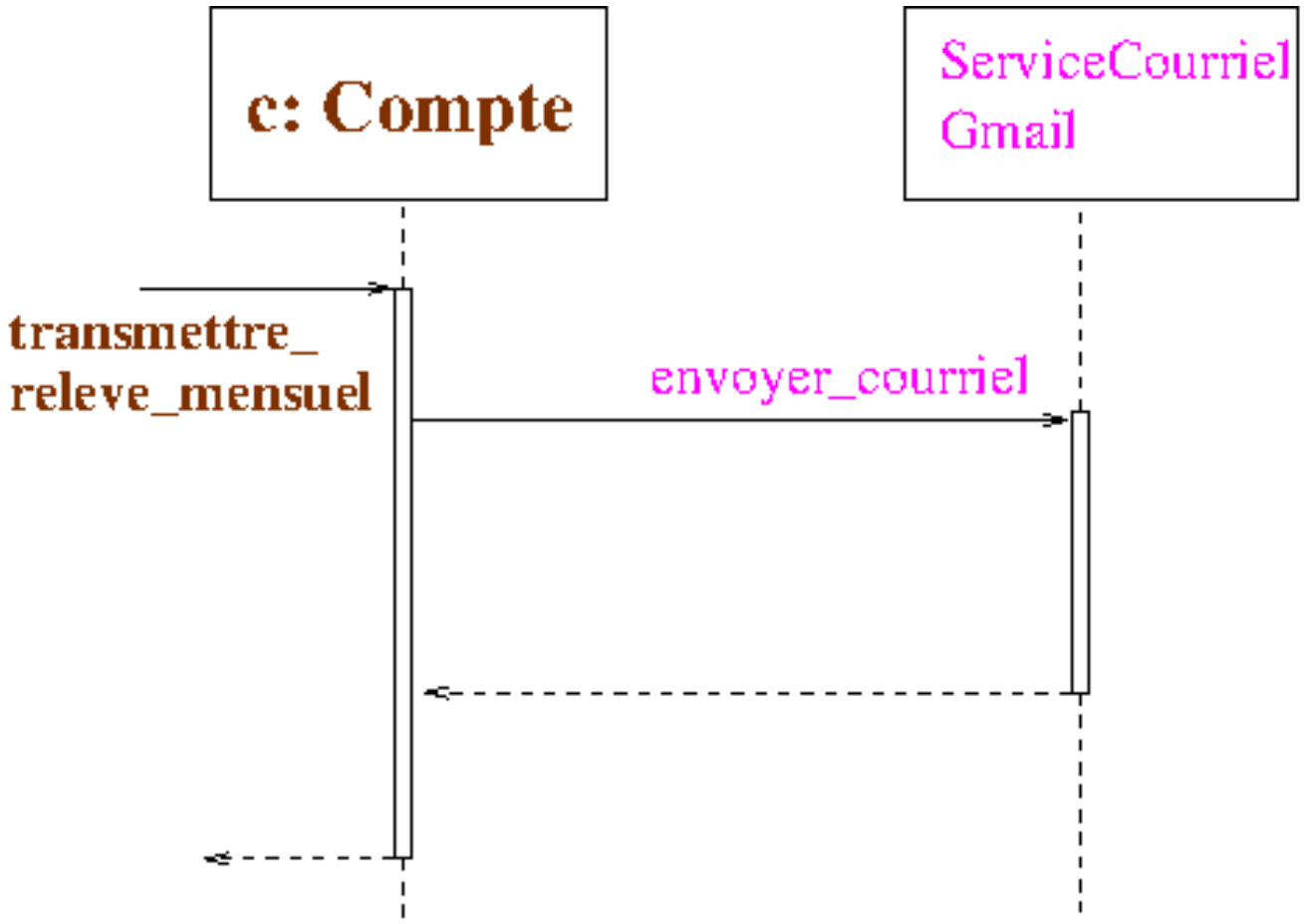
```ruby
class Compte
  attr_reader :client, :solde, :courriel

  def initialize(client, solde_init, courriel = nil)
    @client = client
    @solde = solde_init
    @courriel = courriel
  end

  def deposer( montant ); ...; end

  def retirer( montant ); ...; end

  def transmettre_releve_mensuel( releve )
    ...
  end
end
```

Un diagramme de classe UML pour la classe `Compte`



Exemple : Envoi de relevés mensuels par courriel

Supposons une mise en oeuvre de `transmettre_releve_mensuel` qui utilise le service de courriel de `Gmail`



On veut tester la méthode `transmettre_releve_mensuel` de la classe `Compte`...

Exemple : Envoi de relevés mensuels par courriel

Mise en oeuvre de la méthode `transmettre_releve_mensuel`

```
def transmettre_releve_mensuel( releve )
  msg = <<-EOM
    Cher #{@client}.

    Vous trouverez ci-joint votre releve mensuel.

    Salutations.
  EOM

  ServiceCourrielGmail.
    envoyer_courriel( @courriel,
                      "Releve mensuel",
                      msg + releve )
end
```

Exemple : Envoi de relevés mensuels par courriel

Un service externe pour l'envoi de courriels

```ruby
module ServiceCourrielGmail

  def self.envoyer_courriel( destinataire,
                             sujet,
                             contenu )
  # Source: http://thinkingeek.com/2012/07/29/
  #                 sending-emails-google-mail-ruby
    ...
    Net::SMTP.enable_tls(OpenSSL::SSL::VERIFY_NONE)
    Net::SMTP.start( 'smtp.gmail.com' ... ) do |smtp|
      smtp.send_message( ... )
    end
  end

end
```

Question : Peut-on tester `transmettre_releve_mensuel` ? Si oui, de quelle façon ?

```
def transmettre_releve_mensuel( releve )
  msg = <<-EOM
    Cher #{@client}.

    Vous trouverez ci-joint votre releve mensuel.

    Salutations.
  EOM

  ServiceCourrielGmail.
    envoyer_courriel( @courriel,
                      "Releve mensuel",
                      msg + releve )
end
```

Question : Que se passe-t-il si, dans un test, on utilise directement `ServiceCourrielGmail` ?

Question : Que se passe-t-il si, dans un test, on utilise directement `ServiceCourrielGmail` ?

- Le test pour `transmettre_releve_mensuel` peut être long à exécuter ☹

Question : Que se passe-t-il si, dans un test, on utilise directement `ServiceCourrielGmail` ?

- Le test pour `transmettre_releve_mensuel` peut être long à exécuter ☹
- De «vrais courriels» seront envoyés à chaque fois que les tests seront exécutés ☹

Question : Que se passe-t-il si, dans un test, on utilise directement `ServiceCourrielGmail` ?

- Le test pour `transmettre_releve_mensuel` peut être long à exécuter ☹

- De «vrais courriels» seront envoyés à chaque fois que les tests seront exécutés ☹

⇒ La seule façon de vérifier le résultat du test est . . .

Question : Que se passe-t-il si, dans un test, on utilise directement `ServiceCourrielGmail` ?

- Le test pour `transmettre_releve_mensuel` peut être long à exécuter ☹

- De «vrais courriels» seront envoyés à chaque fois que les tests seront exécutés ☹

⇒ La seule façon de vérifier le résultat du test est . . . d'aller voir dans la boite de courriels ☹

⇒ Le test n'est pas «automatique»

Le problème = Dépendance entre les deux classes ☹

- Le test (et le code !) pour la méthode `transmettre_releve_mensuel` n'est pas indépendant, parce que `Compte` n'est pas indépendant de `ServiceCourrielGmail` ☹

Le problème = Dépendance entre les deux classes ☹

- Le test (et le code !) pour la méthode `transmettre_releve_mensuel` n'est pas indépendant, parce que `Compte` n'est pas indépendant de `ServiceCourrielGmail` ☹

⇒ Il faut «briser» la dépendance qui existe entre `Compte` et `ServiceCourrielGmail` !

Dependency Inversion Principle (DIP)

A. *High-level modules should not depend on low-level modules. Both should depend on abstractions.*

B. *Abstractions should not depend on details. Details should depend on abstractions.*

Robert C. Martin (dit «Uncle Bob»)

Source: *«Agile Software Development—Principles, Patterns, and Practices»*

Exemple : Envoi de relevés mensuels par courriel

Mise en oeuvre de la méthode `transmettre_releve_mensuel`

```
def transmettre_releve_mensuel( releve )
  msg = <<-EOM
    Cher #{@client}.

    Vous trouverez ci-joint votre releve mensuel.

    Salutations.
  EOM

  ServiceCourrielGmail.
    envoyer_courriel( @courriel,
                      "Releve mensuel",
                      msg + releve )
end
```

⇒ Ne respecte pas le DIP ☹

Injection de dépendances

A dependency is an object that can be used (a service).

An injection is the passing of a dependency to a dependent object (a client) that would use it. The service is made part of the client's state.

Source: `https://en.wikipedia.org/wiki/Dependency_injection`

Injection de dépendances. . . plus en détails

Dependency injection is a software design pattern that implements inversion of control for resolving dependencies.

A dependency is an object that can be used (a service). An injection is the passing of a dependency to a dependent object (a client) that would use it. The service is made part of the client's state. [. . .]

Dependency injection allows a program design to follow the dependency inversion principle. The client delegates to external code (the injector) the responsibility of providing its dependencies.

Source: `https://en.wikipedia.org/wiki/Dependency_injection`

Différentes formes d'injection de dépendances

Constructor injection

Setter injection

Injection via un attribut de la classe

Registre de services

Différentes formes d'injection de dépendances

Constructor injection

Les dépendances requises sont fournies dans le constructeur, lors de la création de l'objet.

```
def initialize( client, solde_initial, courriel,
                service_courriel )
  ...
  @service_courriel = service_courriel
end

def transmettre_releve_mensuel( releve )
  ...
  @service_courriel.envoyer_courriel( @courriel,
          "Releve mensuel", msg + releve )
end
```

Différentes formes d'injection de dépendances

Setter injection

L'objet possède une méthode qui permet de spécifier la dépendance.

```
attr_accessor :service_courriel

def transmettre_releve_mensuel( releve )
  ...
  service_courriel.
    envoyer_courriel( @courriel,
                      "Releve mensuel",
                      msg + releve )
end
```

Différentes formes d'injection de dépendances

Injection via un attribut de la classe

La classe possède une méthode qui permet de spécifier la dépendance, applicable à toutes les instances de la classe.

```
class Compte
  # Attributs de classe:
  #   Compte.service_courriel
  #   Compte.service_courriel = service
  class << self
    attr_accessor :service_courriel # Attribut de classe.
  end

  def transmettre_releve_mensuel( releve )
    ...
    Compte.service_courriel.envoyer_courriel(
       @courriel, "Releve mensuel", msg + releve )
  end
end
```

Différentes formes d'injection de dépendances ⋆

Registre de services

Un module gère la liste des services disponibles.

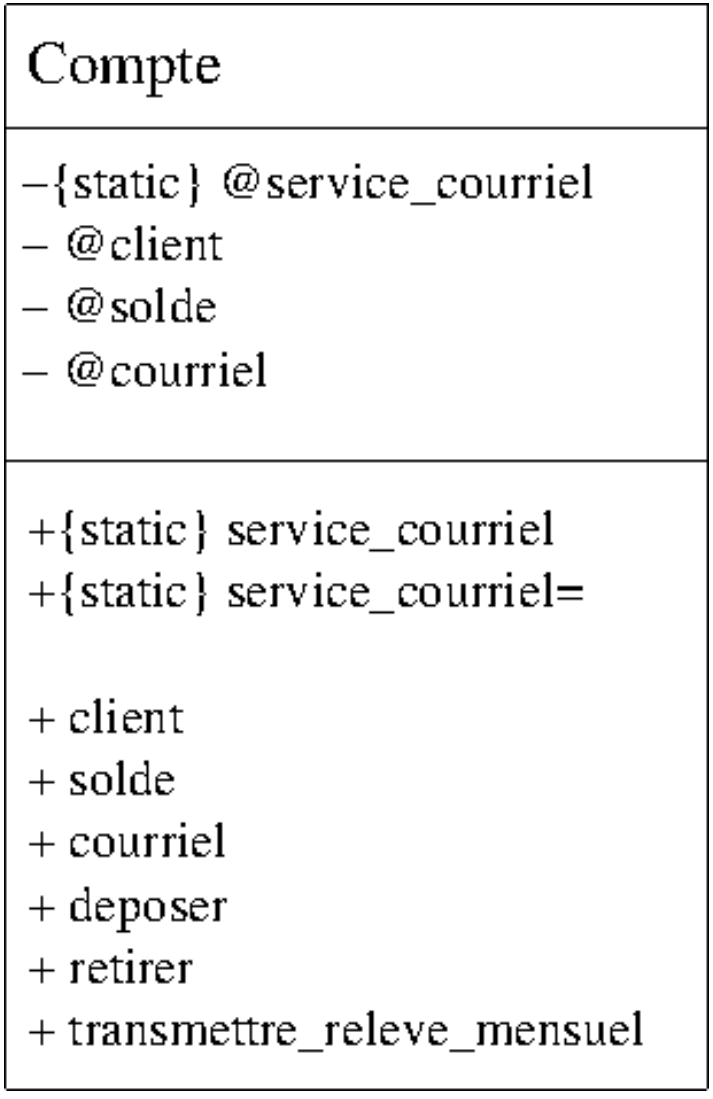
Note : Parfois aussi appelé «*service locator*».

```
class ServicesExternes
  class << self;
    attr_accessor :courriel
  end
end

class Compte
  def transmettre_releve_mensuel( releve )
    ...
    ServicesExternes.courriel.envoyer_courriel(
       @courriel, "Releve mensuel", msg + releve )
  end
end
```

Utilisation d’un attribut de classe pour le compte bancaire

Un diagramme de classe UML pour la classe `Compte` avec attribut de classe `service_courriel`

L'attribut sert pour l'injection du service de courriel et son utilisation dans la classe



Exemple : Envoi de courriels avec injection via un attribut de classe

Comment on effectue l'injection de la dépendance avec l'attribut de classe

```
# On specifie le service approprie a utiliser.
Compte.service_courriel = ServiceCourrielGmail

...

# On cree des objets Compte.
c = Compte.new( 'Guy T.',
                100.00,
                'tremblay.guy@uqam.ca' )
...

# On transmet des releves mensuels.
releve = ...
c.transmettre_releve_mensuel( releve )
...
```

Exemple : Envoi de courriels avec injection via un attribut de classe

Comment on met en oeuvre la méthode `transmettre_releve_mensuel`

```
  def transmettre_releve_mensuel( releve )
    msg = <<-EOM
      Cher #{@client}.

      Vous trouverez ci-joint votre releve mensuel.

      Salutations.
    EOM

    Compte.service_courriel.
      envoyer_courriel( @courriel,
                        "Releve mensuel",
                        msg + releve )
  end
end
```

Mais comment l'injection de dépendances nous aide-t-elle pour les tests unitaires ?

5. Les doublures de tests

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

Test double

A Test Double is an object that stands in for another object in your system during a code example.

Terminologie de G. Meszaros *(«xUnit Test Patterns—Refactoring Test Code»)*

Source: `https://github.com/rspec/rspec-mocks`

Clarification sur le terme «*double*»

Note : Le terme anglais «*double*» est utilisé au sens de «*stunt double*», i.e., au sens du terme français «doublure».

doublure, nom féminin

- . . .
- Prête-nom ou personne qui joue abusivement le rôle d'une autre.
- Acteur secondaire qui se tient prêt à remplacer le titulaire du rôle en cas de besoin.
- Spécialiste (cascadeur) remplaçant effectivement un acteur pour des scènes exceptionnelles et dangereuses.

Source: `http://www.larousse.fr/dictionnaires/francais/doublure/26596`

Différentes sortes de doublures de tests

- 5.1 *Stubs*
- 5.2 *Mocks*
- 5.3 *Autres sortes de doublures de test*
  - *Dummy objects*
  - *Fake objects*
  - *Spies*

Source: `http://www.martinfowler.com/bliki/TestDouble.html`

5.1 Les *stubs*

Exemple : On veut tester un composant dont le comportement dépend de la date courante

Remise d'un travail avec date limite dans l'outil de correction Oto

On crée une boite de remise avec date limite (13/10/2016, 18h00)

```
date_limite = Time.new( 2016, 10, 13, 18, 0, 0 )
boite_remise = Boite.new :INF7235, 'Projet 1', date_limite
```

On remet un travail dans la boite de remise

```
remise_effectuee = boite_remise.rendre_tp @tp, @equipe
```

```
si la date de remise est avant la date_limite
  => remise_effectuee == true

si la date de remise est après la date_limite
  => remise_effectuee == false
```

Exemple : On veut tester un composant dont le comportement dépend de la date courante

Remise d'un travail avec date limite dans l'outil de correction Oto

On crée une boite de remise avec date limite (13/10/2016, 18h00)

```
date_limite = Time.new( 2016, 10, 13, 18, 0, 0 )
boite_remise = Boite.new :INF7235, 'Projet 1', date_limite
```

On remet un travail dans la boite de remise

```
remise_effectuee = boite_remise.rendre_tp @tp, @equipe
```

```
si la date de remise est avant la date_limite
  => remise_effectuee == true

si la date de remise est après la date_limite
  => remise_effectuee == false
```

Question : Comment peut-on tester ce composant ?

Exemple : On veut tester un composant dont le comportement dépend de la date courante

Une solution possible

Un test avec un travail remis avant la date limite

```
date_limite = Time.now + 10
boite_remise = Boite.new :INF7235, 'Projet 1', date_limite
assert boite_remise.rendre_tp @tp, @equipe
```

Un test avec un travail remis après la date limite

```
date_limite = Time.now - 10
boite_remise = Boite.new :INF7235, 'Projet 1', date_limite
refute boite_remise.rendre_tp @tp, @equipe
```

Exemple : On veut tester un composant dont le comportement dépend de la date courante

Une solution possible

Un test avec un travail remis avant la date limite

```
date_limite = Time.now + 10
boite_remise = Boite.new :INF7235, 'Projet 1', date_limite
assert boite_remise.rendre_tp @tp, @equipe
```

Un test avec un travail remis après la date limite

```
date_limite = Time.now - 10
boite_remise = Boite.new :INF7235, 'Projet 1', date_limite
refute boite_remise.rendre_tp @tp, @equipe
```

Question : Quels sont les désavantages de cette solution ?

Exemple : On veut tester un composant dont le comportement dépend de la date courante

Une solution possible

Un test avec un travail remis avant la date limite

```
date_limite = Time.now + 10
boite_remise = Boite.new :INF7235, 'Projet 1', date_limite
assert boite_remise.rendre_tp @tp, @equipe
```

Un test avec un travail remis après la date limite

```
date_limite = Time.now - 10
boite_remise = Boite.new :INF7235, 'Projet 1', date_limite
refute boite_remise.rendre_tp @tp, @equipe
```

Question : Quels sont les désavantages de cette solution ?

- Requiert des appels à `Time` — explicite (date limite dans le test) et implicite (`rendre_tp`)

Exemple : On veut tester un composant dont le comportement dépend de la date courante

Une solution possible

Un test avec un travail remis avant la date limite

```
date_limite = Time.now + 10
boite_remise = Boite.new :INF7235, 'Projet 1', date_limite
assert boite_remise.rendre_tp @tp, @equipe
```

Un test avec un travail remis après la date limite

```
date_limite = Time.now - 10
boite_remise = Boite.new :INF7235, 'Projet 1', date_limite
refute boite_remise.rendre_tp @tp, @equipe
```

Question : Quels sont les désavantages de cette solution ?

- Requiert des appels à `Time` — explicite (date limite dans le test) et implicite (`rendre_tp`)
- Ne peut pas tester des conditions limites — minuit, remise à la date exacte ou une seconde avant/après, etc. ☹

Différentes sortes de doublures de tests

Outil pour un autre type de solution = *Stubs*

- *Stubs provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.*

Source: `http://www.martinfowler.com/bliki/TestDouble.html`

Exemples MiniTest pour une classe `Point`

Une classe `Point` simple

Possède trois méthodes d'instance (`x`, `y` et `==`) et la méthode de classe `new`

```
class Point
  attr_reader :x, :y

  def initialize( x, y )
    @x, @y = x, y
  end

  def ==( autre )
    return nil unless autre.kind_of? Point

    x == autre.x && y == autre.y
  end
end
```

Des exemples de méthodes *stubs* pour `Point`

On peut stubber une méthode d'instance sans argument

```
p = Point.new( 1, 2 )

assert p.x == 1

# Redefinition temporaire (locale au bloc) de x.
p.stub :x, 99 do
  assert p.x == 99
end

# La methode originale est retablie apres le bloc.
assert p.x == 1
```

Des exemples de méthodes *stubs* pour `Point`

On peut stubber une méthode d'instance avec argument(s)

```
p  = Point.new( 1, 2 )
p1 = Point.new( 3, 4 )

refute p == p1  # assert p != p1

p.stub :==, true do
  assert p == p1
end

refute p == p1  # La vraie methode est retablie.
```

Note : `refute p = assert !p`

Des exemples de méthodes *stubs* pour `Point`

On peut stubber une méthode d'instance avec argument(s) en spécifiant plusieurs possibilités (via une `lambda`)

```
p0 = Point.new( 0, 0 )
p1 = Point.new( 1, 1 )
p2 = Point.new( 2, 3 )

assert p0 != p1 && p0 != p2 && p0 == p0

pseudo_egal = ->(p) { p.equal?(p1) || p.x != p.y }
p0.stub :==, pseudo_egal do
  assert p0 == p1    # p1.equal?(p1)
  assert p0 == p2    # p2.x != p2.y
  assert p0 != p0    # !p0.equal?(p1) && p0.x == p0.y
end

assert p0 != p1 && p0 != p2 && p0 == p0
```

Note : `->(p){...}` = `lambda { |p| ... }`

Des exemples de méthodes *stubs* pour `Point`

On peut stubber une méthode de classe, y compris `new`

```
Point.stub :new, "?" do
  assert Point.new( 10, 20 ) == "?"
end
refute Point.new( 10, 20 ) == "?"
```

Exemple MiniTest pour la classe (prédéfinie) `Date`

Un exemple de méthode *stub* pour une classe prédéfinie : `Date`

Un test qui vérifie qu'un travail est accepté s'il est remis avant la date limite

```
it "accepte si la date limite n'est pas depassee" do
  # Date limite de remise: 22 sept. 2015 a 9h00 AM.
  date_limite = Time.new( 2015, 9, 22, 9, 0, 0 )
  boite_remise = Boite.new :INF600A, 'Projet 1', date_limite

  date_remise = Time.new( 2015, 9, 22, 8, 8, 0 )
  Time.stub :now, date_remise do
    remise_effectuee = boite_remise.rendre_tp @tp, @equipe
    assert remise_effectuee
  end
end
```

Un exemple de méthode *stub* pour une classe prédéfinie : `Date`

Un test qui vérifie qu'un travail est refusé s'il est remis après la date limite

```
it "refuse si la date limite est depassee" do
  # Date limite de remise: 22 sept. 2015 a 9h00 AM.
  date_limite = Time.new( 2015, 9, 22, 9, 0, 0 )
  boite_remise = Boite.new :INF600A, 'Projet 1', date_limite

  date_retard = Time.new( 2015, 9, 22, 11, 11, 0 )
  Time.stub :now, date_retard do
    remise_effectuee = boite_remise.rendre_tp @tp, @equipe
    refute remise_effectuee
  end
end
```

Exemple MiniTest pour `transmettre_releve_mensuel`

Un test pour `transmettre_releve_mensuel` avec un *stub* pour `envoyer_courriel`

La mise en oeuvre de `transmettre_releve_mensuel`

```
def transmettre_releve_mensuel( releve )
  msg = <<-EOM
    Cher #{@client}.

    Vous trouverez ci-joint votre releve mensuel.

    Salutations.
  EOM

  Compte.service_courriel.
    envoyer_courriel( @courriel,
                      "Releve mensuel",
                      msg + releve )
end
```

Un test pour `transmettre_releve_mensuel` avec un *stub* pour `envoyer_courriel`

Un *stub* pour `envoyer_courriel`

Stub pour `envoyer_courriel` = prend note des informations transmises si on l'appelle (via variables non locales), ce qui permet ensuite de vérifier que la méthode a été appelée.

```
# Variables non-locales au bloc du lambda,
# qui seront modifiees si le lambda est appelle.
adresse_utilisee = le_sujet = le_releve = nil

# La lambda-expression qui sera utilisee comme stub.
envoyer_courriel_bidon = lambda do |c, s, r|
  adresse_utilisee, le_sujet, le_releve = c, s, r
end
```

Un test pour `transmettre_releve_mensuel` avec un *stub* pour `envoyer_courriel`

Un test qui utilise le *stub* en vérifiant que la méthode `envoyer_courriel` a bien été

```
# On definit le service a utiliser via l'attribut de c
Compte.service_courriel = ServiceCourrielGmail

# On effectue un appel a transmettre_releve_mensuel...
# mais en utilisant le stub pour envoyer_courriel
releve_a_envoyer = "...un releve bidon..."
ServiceCourrielGmail.stub :envoyer_courriel,
                          envoyer_courriel_bidon do
  @c.transmettre_releve_mensuel(releve_a_envoyer)
end


# On verifie que l'appel a envoyer_courriel a bien eu
adresse_utilisee.must_equal @c.courriel
le_sujet.must_equal "Releve mensuel"
le_releve.must_match /#{releve_a_envoyer}/
```

Les *stubs* sont utiles, mais complexes à utiliser lorsqu'on veut vérifier des contraintes sur les appels

- Dans l'exemple de `transmettre_releve_mensuel`, on a pu utiliser une méthode *stub*, mais son utilisation est commplexe, car on a certaines attentes — certaines «*expectations*» — quant à ce qui doit se passer, et avec un *stub* simple, c'est compliqué à tester (modification de variables non-locales, tests sur les variables modifiées, etc.).

- Objets *mocks* = autre forme de doublure de test avec méthodes *stubs*... mais pour lesquels on a la possibilité de spécifier des contraintes sur ce qui est attendu !

Traduction possible de «*mock*» = «simulacre»

5.2 Les *mocks*

Question : Comment fait-on pour tester `Compte.transmettre_releve_mensuel` ?

? ?

Un premier essai pour `transmettre_releve_mensuel`

Dépend de `ServiceCourrielGmail`!?

```
it "transmet le courriel avec le bon message" do
  Compte.service_courriel = ServiceCourrielGmail

  @c.transmettre_releve_mensuel(releve_a_envoyer)

  assert ?????
end
```

Un premier essai pour `transmettre_releve_mensuel`

Dépend de `ServiceCourrielGmail` !?

```
it "transmet le courriel avec le bon message" do
  Compte.service_courriel = ServiceCourrielGmail

  @c.transmettre_releve_mensuel(releve_a_envoyer)

  assert ?????
end
```

Question : Comment peut-on vérifier — sans aller voir dans les «vrais» courriels — que `transmettre_releve_courriel` a bien fait son travail ?

Un premier essai pour `transmettre_releve_mensuel`

Dépend de `ServiceCourrielGmail` !?

```
it "transmet le courriel avec le bon message" do
  Compte.service_courriel = ServiceCourrielGmail

  @c.transmettre_releve_mensuel(releve_a_envoyer)

  assert ?????
end
```

Question : Comment peut-on vérifier — sans aller voir dans les «vrais» courriels — que `transmettre_releve_courriel` a bien fait son travail ?

Il faudrait pouvoir analyser/vérifier ce qu'a fait `ServiceCourrielGMail...`

Un deuxième essai avec un *stub* pour `transmettre_releve_mensuel`

Le test semble dépendre de `ServiceCourrielGmail`, donc on le *stubbe* ☹

```
it "transmet le courriel avec le bon message" do
  Compte.service_courriel = ServiceCourrielGmail

  envoyer_courriel_bidon = ...
  ServiceCourrielGmail.stub :envoyer_courriel,
                            envoyer_courriel_bidon do
    @c.transmettre_releve_mensuel(releve_a_envoyer)
  end

  assert ?????
end
```

Le *stub* peut prendre en note certaines informations, qu'on vérifie ensuite... mais c'est un peu compliqué ☹

Solution alternative = utiliser un objet *mock* !

Qu'est-ce qu'un objet *mock* ?

Terminologie de G. Meszaros *(«xUnit Test Patterns—Refactoring Test Code»)*

Test double

A Test Double is an object that stands in for another object in your system during a code example.

Mock object

a Mock Object is a Test Double that supports message expectations **and** *method stubs.*

Source: `https://github.com/rspec/rspec-mocks`

Qu'est-ce qu'un objet *mock* ?

Mocks are **pre-programmed with expectations** *which form a specification of the calls they are expected to receive.*

They can throw an exception if they receive a call they don't expect and *are checked during verification to ensure they got all the calls they were expecting*.

Source: `http://www.martinfowler.com/bliki/TestDouble.html`

Exemples de *mocks* en MiniTest

On peut spécifier des méthodes *stubs* sur un *mock*

Exemple tiré de la documentation de `MiniTest` spécifiant une méthode `stub`

```
mock = MiniTest::Mock.new

# Si un appel a :meaning_of_life est recu (sans arg.),
# alors on retourne 42.
mock.expect(:meaning_of_life, 42)

mock.meaning_of_life # => 42
```

On peut spécifier des méthodes *stubs* sur un *mock*

Autre exemple tiré de la documentation de `MiniTest` spécifiant une méthode `stub`

```
mock = MiniTest::Mock.new

# Si un appel :do_something_with(some_object, true)
# est recu, alors on retourne true.
some_object = Object.class
mock.expect(:do_something_with, true, [some_obj, true])

mock.do_something_with(some_obj, true) # => true
```

On peut spécifier des méthodes *stubs* sur un *mock*

Exemple illustrant une méthode *stub* générique

```
mock = MiniTest::Mock.new

# Si on fait a un appel a foo(_, num1, "abc")
#   où num1 est un nombre quelconque
# alors on retourne 42
mock.expect(:foo, 42, [Object, Fixnum, "abc"])

mock.foo(Object.new, 0, "abc") # => 42
```

Les arguments sont *matchés* avec «===» et non avec «==».

On peut vérifier que les appels *stubbés* ont effectivement eu lieu

L'appel attendu est effectué

```
it "appelle foo" do
  mock = MiniTest::Mock.new
  mock.expect( :foo, 42 )

  mock.foo  # => 42

  mock.verify
end
```

```
# Running:

.
[...]
1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
```

On peut vérifier que les appels *stubbés* ont effectivement eu lieu

L'appel attendu n'est pas effectué

```
it "n'appelle pas foo" do
  mock = MiniTest::Mock.new
  mock.expect( :foo, 42 )

  mock.verify
end
```

```
# Running:
E
[...]
  1) Error:
Mocks#test_0001_n'appelle pas foo:
MockExpectationError: expected foo() => 42, got []
    mocks.rb:21:in 'block (2 levels) in <main>'

1 runs, 0 assertions, 0 failures, 1 errors, 0 skips
```

On peut vérifier que les appels *stubbés* ont effectivement eu lieu

Un appel est fait mais avec les mauvais arguments

```
it "appelle foo avec les mauvais arguments" do
  mock = MiniTest::Mock.new
  mock.expect( :foo, 42, [0, Symbol] )

  mock.foo( 10, :abc )
end
```

```
# Running:
E
[...]
  1) Error:
Mocks#test_0003_appelle foo avec les mauvais arguments:
MockExpectationError: mocked method :foo called with\
                          unexpected arguments [10, :abc]
    mocks.rb:27:in 'block (2 levels) in <main>'

1 runs, 0 assertions, 0 failures, 1 errors, 0 skips
```

Exemple de *mock* pour tester `transmettre_releve_mensuel`

Un test pour `transmettre_releve_mensuel` qui utilise un *mock*

Grâce à l'injection de dépendance, pas besoin d'utiliser `ServiceCourrielGmail` !

```
it "transmet le courriel avec le bon message" do
  releve = "...un releve bidon..."

  mock_courriel = MiniTest::Mock.new
  mock_courriel.expect( :envoyer_courriel,
                        nil,
                        [@c.courriel,
                         "Releve mensuel",
                         /#{@c.client}.*#{releve}/m] )

  Compte.service_courriel = mock_courriel
  @c.transmettre_releve_mensuel( releve )

  mock_courriel.verify
end
```

La méthode `transmettre_releve_mensuel...` ne sait pas si l'objet reçu est *mock* ou un vrai service !

```
def transmettre_releve_mensuel( releve )
  msg = <<-EOM
    Cher #{@client}.

    Vous trouverez ci-joint votre releve mensuel.

    Salutations.
  EOM

  Compte.service_courriel.
    envoyer_courriel( @courriel,
                      "Releve mensuel",
                      msg + releve )
end
```

5.3 Autres sortes de doublures de test

Différentes sortes de doublures de tests

Terminologie de G. Meszaros *(«xUnit Test Patterns—Refactoring Test Code»)* :

> *A Test Double is an object that stands in for another object in your system during a code example.*

Différentes sortes de doublures :

- *Stubs*
- *Mocks*
- *Dummy objects*
- *Fake objects*
- *Spies*

Source: `http://www.martinfowler.com/bliki/TestDouble.html`

Différentes sortes de doublures de tests

Les *dummies*

- *Dummy objects are passed around but never actually used. Usually they are just used to fill parameter lists.*

Source: `http://www.martinfowler.com/bliki/TestDouble.html`

Un exemple de *dummy object* : `releve`

```
it "transmet le courriel avec le bon message" do
  releve = String.new

  mock_courriel = MiniTest::Mock.new
  mock_courriel.expect( :envoyer_courriel,
                        nil,
                        [@c.courriel,
                         "Releve mensuel",
                         String] )

  Compte.service_courriel = mock_courriel
  @c.transmettre_releve_mensuel( releve )

  mock_courriel.verify
end
```

Différentes sortes de doublures de tests

Les *fakes*

- *Fake objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an InMemoryTestDatabase is a good example).*

Source:

`http://www.martinfowler.com/bliki/TestDouble.html`

Différentes sortes de doublures de tests

Les *fakes*

- *Fake objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an InMemoryTestDatabase is a good example).*

Source:

`http://www.martinfowler.com/bliki/TestDouble.html`

Différentes sortes de doublures de tests

Les *spies*

- *Spies are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages it was sent.*

Source: `http://www.martinfowler.com/bliki/TestDouble.html`

Un exemple de *spy*

```
gem 'spy'
require 'spy/integration'

c = Compte.new( "Guy T.", 500, "tremblay.guy@uqam.ca" )

deposer_spy = Spy.on( c, :deposer )

c.deposer( 100 )
c.deposer( 250 )

assert deposer_spy.has_been_called?
deposer_spy.calls.count.must_equal 2

premier_appel = deposer_spy.calls.first
premier_appel.object.must_equal c
premier_appel.args.must_equal [100]
```

6. Quelques éléments complémentaires

6.1 Doublures de tests

Indirect Input

When the behavior of the **system under test (SUT)** *is affected by the values returned by another component whose services it uses, we call those values the indirect inputs of the SUT.*

Indirect inputs may consist of actual return values of functions, updated (out) parameters of procedures or subroutines, and any errors of exceptions raised by the **depended-on component (DOC)**.

[. . .]

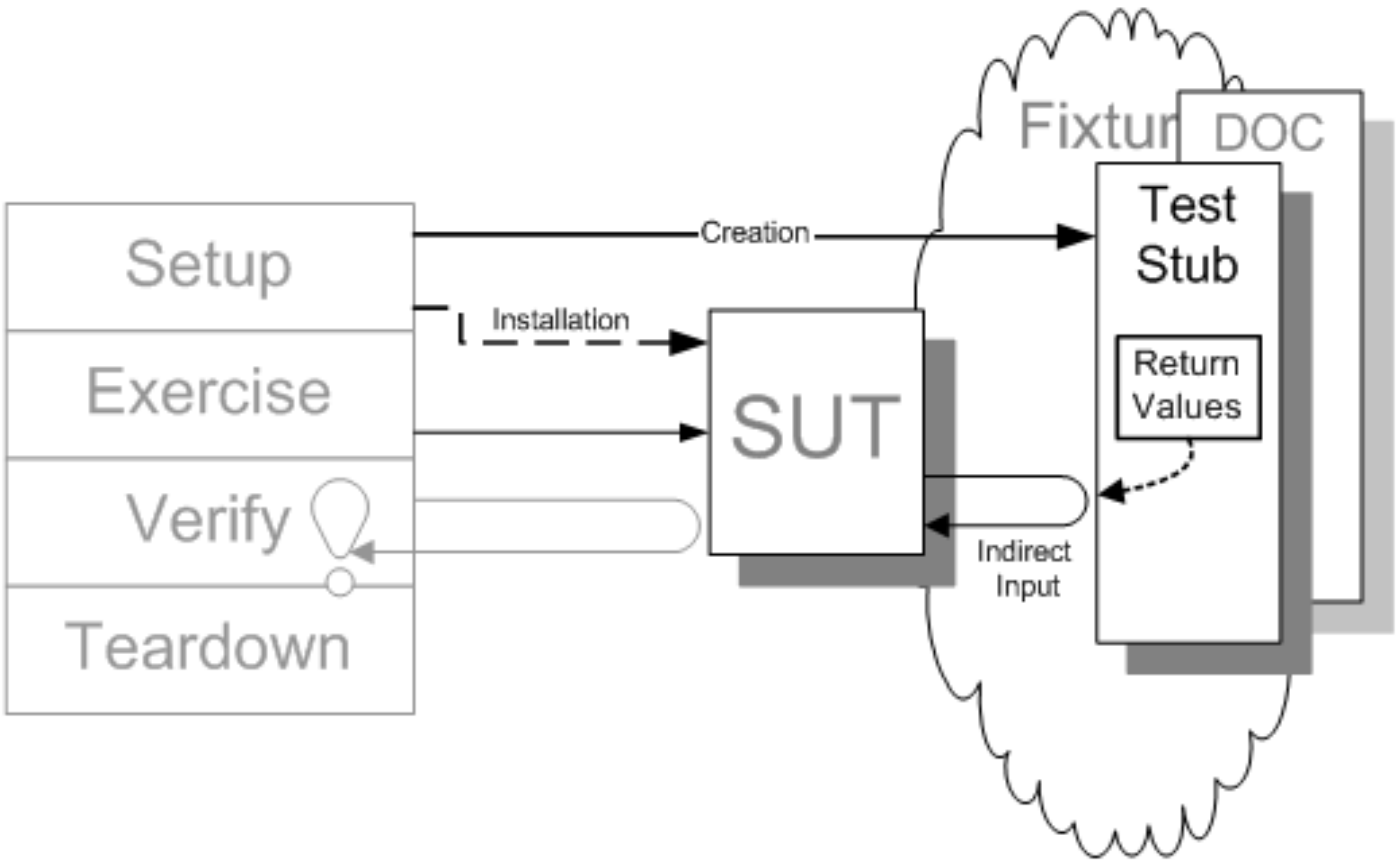
We often use a Test Stub to inject the indirect inputs into the SUT.

Source: `http://xunitpatterns.com/indirect%20input.html`

Test Stub

How can we verify logic independently when it depends on indirect inputs from other software components?

We replace a real object with a test-specific object that feeds the desired indirect inputs into the system under test.



Source: `http://xunitpatterns.com/Test%20Stub.html`

Indirect Output

When the behavior of the **system under test (SUT)** *includes actions that cannot be observed through the public API of the SUT but which are seen or experienced by other systems or application components, we call those actions the indirect outputs of the SUT.*

Indirect outputs may be method or function calls to another component, messages sent on a message channel (e.g. MQ or JMS), records inserted into a database or written to a file.

Verification of the indirect output behaviors of the SUT requires the appropriate observation points on the "back side" of the SUT.

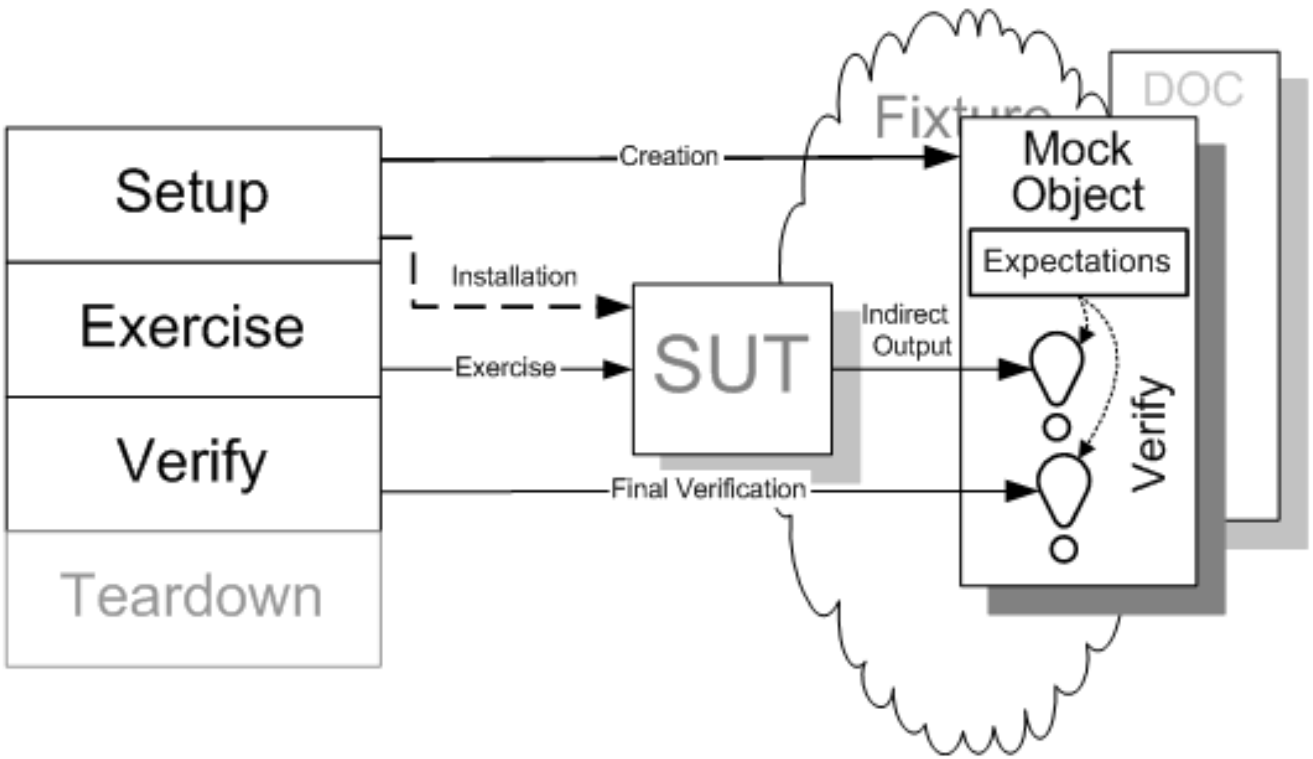
Mock Objects are often used to implement the observation point by intercepting the indirect outputs of the SUT and comparing them to the expected values.

Source: `http://xunitpatterns.com/indirect%20output.html`

Mock Object

How do we implement ***Behavior Verification*** *for indirect outputs of the SUT?*
How can we verify logic independently when it depends on indirect inputs from other software components?

Replace an object the system under test (SUT) depends on with a test-specific object that verifies it is being used correctly by the SUT

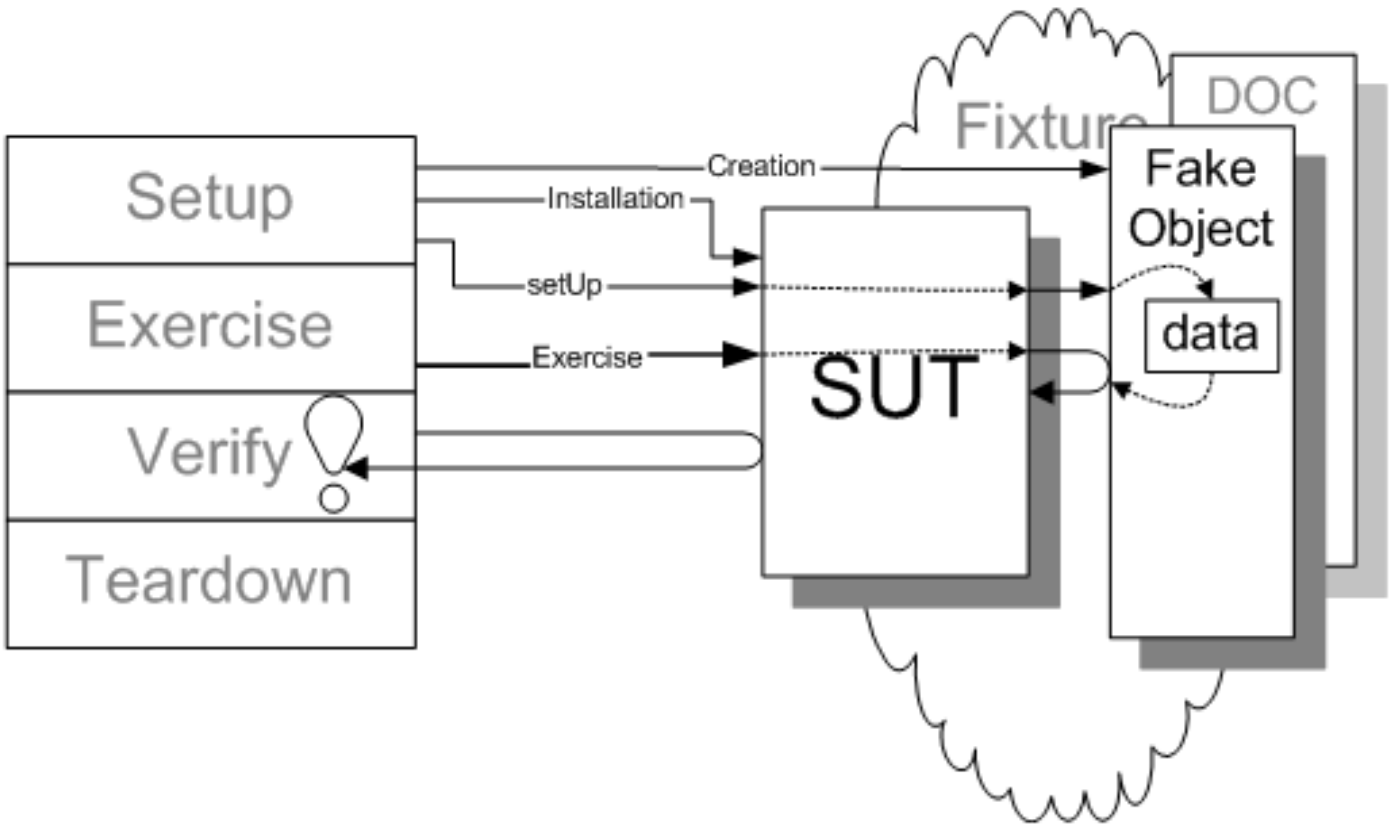


Source: `http://xunitpatterns.com/Mock%20Object.html`

Fake Object

How can we verify logic independently when depended-on objects cannot be used?
*How can we avoid **Slow Tests**?*

Replace a component that the system under test (SUT) depends on with a much lighter-weight implementation.



Source: `http://xunitpatterns.com/Fake%20Object.html`

6.2 Organisation des tests

Les tests peuvent être organisés de différentes façons

- **Testcase Class per Class** *: We put all the Test Methods for one system under test (SUT) class onto a single Testcase Class.*
- **Testcase Class per Feature** *: We group the Test Methods onto Testcase Classes based on which testable feature of the SUT they exercise.*
- **Testcase Class per Fixture** *: We organize Test Methods into Testcase Classes based on commonality of the test fixture.*

Source: `http://xunitpatterns.com/Testcase%20Class%20Structure.html`

Les tests peuvent être organisés de différentes façons

- **Testcase Class per Class** *: We put all the Test Methods for one system under test (SUT) class onto a single Testcase Class.*
- **Testcase Class per Feature** *: We group the Test Methods onto Testcase Classes based on which testable feature of the SUT they exercise.*
- **Testcase Class per Fixture** *: We organize Test Methods into Testcase Classes based on commonality of the test fixture.*

Source: `http://xunitpatterns.com/Testcase%20Class%20Structure.html`

Les tests peuvent être organisés de différentes façons

- **Testcase Class per Class** *: We put all the Test Methods for one system under test (SUT) class onto a single Testcase Class.*
- **Testcase Class per Feature** *: We group the Test Methods onto Testcase Classes based on which testable feature of the SUT they exercise.*
- **Testcase Class per Fixture** *: We organize Test Methods into Testcase Classes based on commonality of the test fixture.*

Source: `http://xunitpatterns.com/Testcase%20Class%20Structure.html`

Les tests peuvent être organisés de différentes façons

- **Testcase Class per Class** *: We put all the Test Methods for one system under test (SUT) class onto a single Testcase Class.*
- **Testcase Class per Feature** *: We group the Test Methods onto Testcase Classes based on which testable feature of the SUT they exercise.*
- **Testcase Class per Fixture** *: We organize Test Methods into Testcase Classes based on commonality of the test fixture.*

  In xUnit, a **test fixture** *is all the things we need to have in place in order to run a test and expect a particular outcome. Some people call this the test context.*

Source: `http://xunitpatterns.com/Testcase%20Class%20Structure.html`

6.3 *Result Verification Patterns*

Les deux principales stratégies pour vérifier les résultats d'un test

Donc pour effectuer la troisième étape (*Verify*) d'un *Four-Phase Test*

State Verification

Behavior Verification

Les deux principales stratégies pour vérifier les résultats d'un test

Donc pour effectuer la troisième étape (*Verify*) d'un *Four-Phase Test*

State Verification

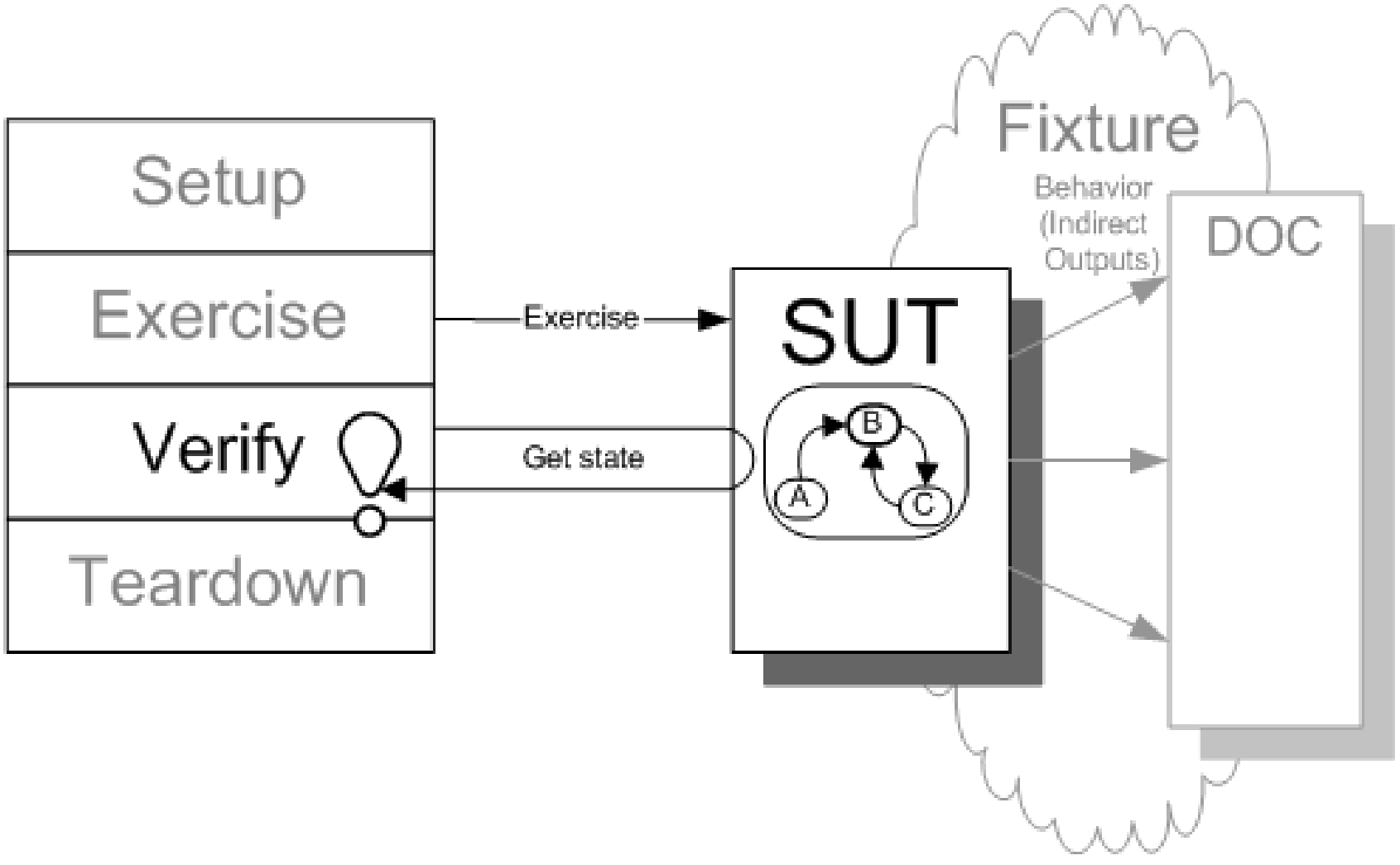
«we determine whether the exercised method worked correctly by examining the state of the SUT and its collaborators after the method was exercised.»

Source: *M. Fowler,* «Mocks Aren't Stubs», `http://martinfowler.com/articles/mocksArentStubs.html`

Behavior Verification

State Verification

Source : `http://xunitpatterns.com/State%20Verification.html`



State Verification

How do we make tests self-checking when there is state to be verified ?

We inspect the state of the system under test (SUT) after it has been exercised and compare it to the expected state.

How it works

We exercise the SUT by invoking the methods of interest. Then, as a separate step, we interact with the SUT to retrieve its post-exercise state and compare this with the expected end state by calling Assertion Methods.

Normally, we can access the state of the SUT simply by calling methods or functions that return its state. [. . .]

Source: `http://xunitpatterns.com/State%20Verification.html`

Les deux principales stratégies pour vérifier les résultats d'un test

Donc pour effectuer la troisième étape (*Verify*) d'un *Four-Phase Test*

State Verification

Behavior Verification

Behavior verification describes a style of testing where the execution of a method is expected to generate specific interactions between objects.

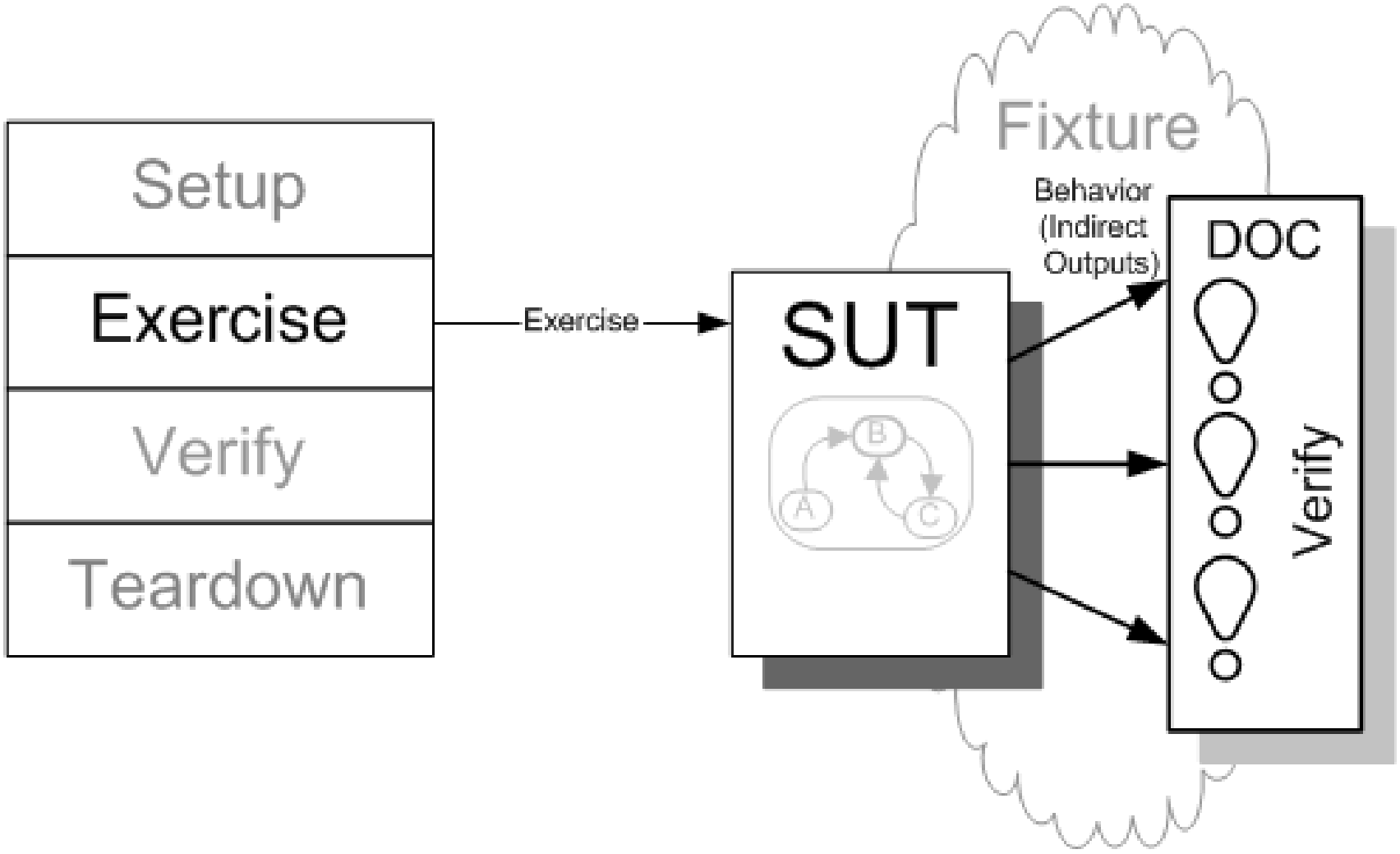
[. . .]

Behavior specification is about specifying how the system should behave as it operates rather than specifying the expected end state of the system after it has completed an operation.

Source*: J. Fields,* «Working Effectively with Unit Tests»

Behavior Verification

Source : `http://xunitpatterns.com/Behavior%20Verification.html`



Behavior Verification

How do we make tests self-checking when there is no state to verify ?

We capture the indirect outputs of the SUT as they occur and compare them to the expected behavior.

How it works

Each test specifies not only how the client of the SUT interacts with it during the exercise SUT phase of the test but also how the SUT should interact with the components on which it should depend. This ensures that the SUT really is behaving as specified rather than just ending up in the correct post-exercise state.

Behavior Verification almost always involves interacting with or replacing a depended-on component (DOC) that the SUT interacts with at run time.

Source: `http://xunitpatterns.com/Behavior%20Verification.html`

Un exemple de M. Fowler

Exemple de M. Fowler

http://martinfowler.com/articles/mocksArentStubs.html

Description du problème

We want to take an order object and fill it from a warehouse object. The order is very simple, with only one product and a quantity. The warehouse holds inventories of different products.

When we ask an order to fill itself from a warehouse there are two possible responses.

- *If there's enough product in the warehouse to fill the order, the order becomes filled and the warehouse's amount of the product is reduced by the appropriate amount.*
- *If there isn't enough product in the warehouse then the order isn't filled and nothing happens in the warehouse.*

Exemple de M. Fowler : *State Verification*

Contexte et `setUp`

```
public class OrderStateTester extends TestCase {
  private static String TALISKER =
    "Talisker";
  private static String HIGHLAND_PARK =
    "Highland Park";
  private Warehouse warehouse =
    new WarehouseImpl();

  protected void setUp() throws Exception {
    warehouse.add(TALISKER, 50);
    warehouse.add(HIGHLAND_PARK, 25);
  }

  ...
}
```

Source: `http://martinfowler.com/articles/mocksArentStubs.html`

Exemple de M. Fowler : *State Verification*

Premier test : Il y a suffisamment d'items en stock

```
public void testOrderIsFilledIfEnoughInWarehouse() {
    // setup
    Order order = new Order(TALISKER, 50);

    // exercise
    order.fill(warehouse);

    // verify
    assertTrue(order.isFilled());
    assertEquals(0, warehouse.getInventory(TALISKER));
}
```

Source: `http://martinfowler.com/articles/mocksArentStubs.html`

Question : Quel type d'injection de dépendances est utilisé dans cet exemple ?

```
private Warehouse warehouse = new WarehouseImpl();

public void testOrderIsFilledIfEnoughInWarehouse() {
  Order order = new Order(TALISKER, 50);
  order.fill(warehouse);
  assertTrue(order.isFilled());
  assertEquals(0, warehouse.getInventory(TALISKER));
}
```

Question : Quel type d'injection de dépendances est utilisé dans cet exemple ?

```
private Warehouse warehouse = new WarehouseImpl();

public void testOrderIsFilledIfEnoughInWarehouse() {
  Order order = new Order(TALISKER, 50);
  order.fill(warehouse);
  assertTrue(order.isFilled());
  assertEquals(0, warehouse.getInventory(TALISKER));
}
```

«Parameter Injection is a form of Dependency Injection in which the SUT does not keep or initialize a reference to the DOC ; instead, it is passed in as an argument of the method being called on the SUT. All clients of the SUT whether they are tests or production code, supply the depended-on component. This makes the SUT more independent of the context since it makes no assumptions about the dependency other than its usage interface. The main drawback is that it forces the client to know about the dependency.» [Mes07]

Exemple de M. Fowler : *State Verification*

Deuxième test : Il n'y a pas assez d'items en stock

```
public void testOrderDoesNotRemoveIfNotEnough() {
  // setup
  Order order = new Order(TALISKER, 51);

  // exercise
  order.fill(warehouse);

  // verify
  assertFalse(order.isFilled());
  assertEquals(50, warehouse.getInventory(TALISKER));
}
```

Source: http://martinfowler.com/articles/mocksArentStubs.html

Exemple de M. Fowler : *Behavior Verification*

Contexte

```
public class OrderInteractionTester
                extends MockObjectTestCase {
  private static String TALISKER =
    "Talisker";

  ...
}
```

Exemple de M. Fowler : *Behavior Verification*

Premier test : Il y a suffisamment d'items en stock

```
public void testFillingRemovesInventoryIfInStock() {
  // setup - data
  Order order = new Order(TALISKER, 50);
  Mock warehouseMock = new Mock(Warehouse.class);

  // setup - expectations
  warehouseMock.expects(once()).method("hasInventory")
    .with(eq(TALISKER),eq(50))
    .will(returnValue(true));
  warehouseMock.expects(once()).method("remove")
    .with(eq(TALISKER), eq(50))
    .after("hasInventory");

  // exercise
  order.fill((Warehouse) warehouseMock.proxy());

  // verify
  warehouseMock.verify();
  assertTrue(order.isFilled());
}
```

Exemple de M. Fowler : *Behavior Verification*

Deuxième test : Il n'y a pas assez d'items en stock

```
public void testFillingDoesNotRemoveIfNotEnoughInStock() {
  // setup - data
  Order order = new Order(TALISKER, 51);
  Mock warehouse = mock(Warehouse.class);

  // setup - expectations
  warehouse.expects(once()).method("hasInventory")
    .withAnyArguments()
    .will(returnValue(false));

  // exercise
  order.fill((Warehouse) warehouse.proxy());

  // verify
  warehouseMock.verify();
  assertFalse(order.isFilled());
}
```

Un autre exemple de M. Fowler illustrant aussi la différence entre *stub* et *mock*

Un *stub* pour un service de courriel

```
public interface MailService {
  public void send(Message msg);
}

public class MailServiceStub implements MailService {
  private List<Message> messages =
     new ArrayList<Message>();

  public void send(Message msg) {
    messages.add(msg);
  }

  public int numberSent() {
    return messages.size();
  }
}
```

Un test avec *stub* et *state verification*

```
class OrderStateTester ...
  public void testOrderSendsMailIfUnfilled() {
    // setup
    Order order = new Order(TALISKER, 51);
    MailServiceStub mailer = new MailServiceStub();
    order.setMailer(mailer);

    // exercise
    order.fill(warehouse);

    // verify
    assertEquals(1, mailer.numberSent());
  }

  ...
```

Un test avec *stub* et *state verification*

```
class OrderStateTester ...
  public void testOrderSendsMailIfUnfilled() {
    // setup
    Order order = new Order(TALISKER, 51);
    MailServiceStub mailer = new MailServiceStub();
    order.setMailer(mailer);

    // exercise
    order.fill(warehouse);

    // verify
    assertEquals(1, mailer.numberSent());
  }

  ...
```

Question : S'agit-il ici vraiment d'un *stub* ?

Un test avec *stub* et *state verification*

```
class OrderStateTester ...
  public void testOrderSendsMailIfUnfilled() {
    // setup
    Order order = new Order(TALISKER, 51);
    MailServiceStub mailer = new MailServiceStub();
    order.setMailer(mailer);

    // exercise
    order.fill(warehouse);

    // verify
    assertEquals(1, mailer.numberSent());
  }

  ...
```

Question : S'agit-il ici vraiment d'un *stub* ?

En fait, c'est un *spy*, mais selon Fowler «*spies are stubs that also record*

Un test avec *mock* et *behavior verification*

```
class OrderInteractionTester ...
  public void testOrderSendsMailIfUnfilled() {
    // setup
    Order order = new Order(TALISKER, 51);
    Mock warehouse = mock(Warehouse.class);
    Mock mailer = mock(MailService.class);
    order.setMailer((MailService) mailer.proxy());
    mailer.expects(once()).method("send");
    warehouse.expects(once()).method("hasInventory")
      .withAnyArguments()
      .will(returnValue(false));

    // exercise
    order.fill((Warehouse) warehouse.proxy());

    // verify
    warehouse.verify();
  }
}
```

Stateist vs. Mockist Testing

State/Behavior Verification vs. *Stubs/Mocks*

Typiquement :

Stubs/Spies	$\Leftrightarrow$	*State Verification*
Mocks	$\Leftrightarrow$	*Behavior Verification*

State/Behavior Verification vs. *Stubs/Mocks*

Typiquement :

Stubs/Spies	$\Leftrightarrow$	*State Verification*	*Stateist testing*
Mocks	$\Leftrightarrow$	*Behavior Verification*	*Mockist testing*

Les deux principales stratégies pour concevoir les tests

Stateist Testing

«A stateist tester asserts that a method returns a particular value.»

Mockist Testing

«A mockist tester asserts that a method triggers a specific set of interactions with the object's dependencies.

[. . .]

A mockist draws a thick line between unit test and functional or integration tests. For a mockist, a unit test must only test a single unit. Test doubles replace any and all dependencies, ensuring that only an error in the object under test will cause a failure.»

Source: «*On Mocks and Mockist Testing*», J. Golick,

`http://jamesgolick.com/2010/3/10/on-mocks-and-mockist-testing.html`

6.4 Quelques *Test Smells*

Qu'est-ce qu'un *code smell* ?

Code smell, also known as bad smell, in computer programming code, refers to any symptom in the source code of a program that possibly indicates a deeper problem.

[. . .]

Another way to look at smells is with respect to principles and quality : "smells are certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality".

Code smells are usually not bugs—they are not technically incorrect and do not currently prevent the program from functioning. Instead, they indicate weaknesses in design that may be slowing down development or increasing the risk of bugs or failures in the future.

Source: `https://en.wikipedia.org/wiki/Code_smell`

Meszaros classe les *Test Smells* en trois catégories

Code Smells

Behavior Smells

Project Smells

Meszaros classe les *Test Smells* en trois catégories

Code Smells

- *Obscure Test*
- *Conditional Test Logic*
- *Hard-to-Test Code*
- *Test Code Duplication*
- *Test Logic in Production*

Meszaros classe les *Test Smells* en trois catégories

Behavior Smells

- *Assertion Roulette*
- *Erratic Test*
- *Fragile Test*
- *Frequent Debugging*
- *Manual Intervention*
- *Slow Tests*

Meszaros classe les *Test Smells* en trois catégories

Project Smells

- *Buggy tests*
- *Developer Not Writing Tests*
- *High Test Maintenance Cost*
- *Production Bugs*

Quelques *Behavior Smells*

Assertion Roulette

It is hard to tell which of several assertions within the same test method caused a test failure.

Symptoms

A test fails. Upon examining the output of the Test Runner, we cannot determine exactly which assertion had failed.

Causes

- *Eager Test = A single test verifies too much functionality.*
  Solution = Single-Condition Tests !
- *Missing Assertion Message*

Source: `http://xunitpatterns.com/Assertion%20Roulette.html`

Assertion Roulette et *Single-Condition Tests*

We should avoid the temptation to test as much as functionality as possible in a single Test Method *because that can result in* Obscure Tests*. In fact, it is preferable to have many small* Single-Condition Tests*.*

Source*: «xUnit Test Patterns», p. 359*

Erratic Test

One or more tests are behaving erratically ; sometimes they pass and sometimes they fail.

Symptoms

We have one or more tests that run but give different results depending on when they are run and who is running them.

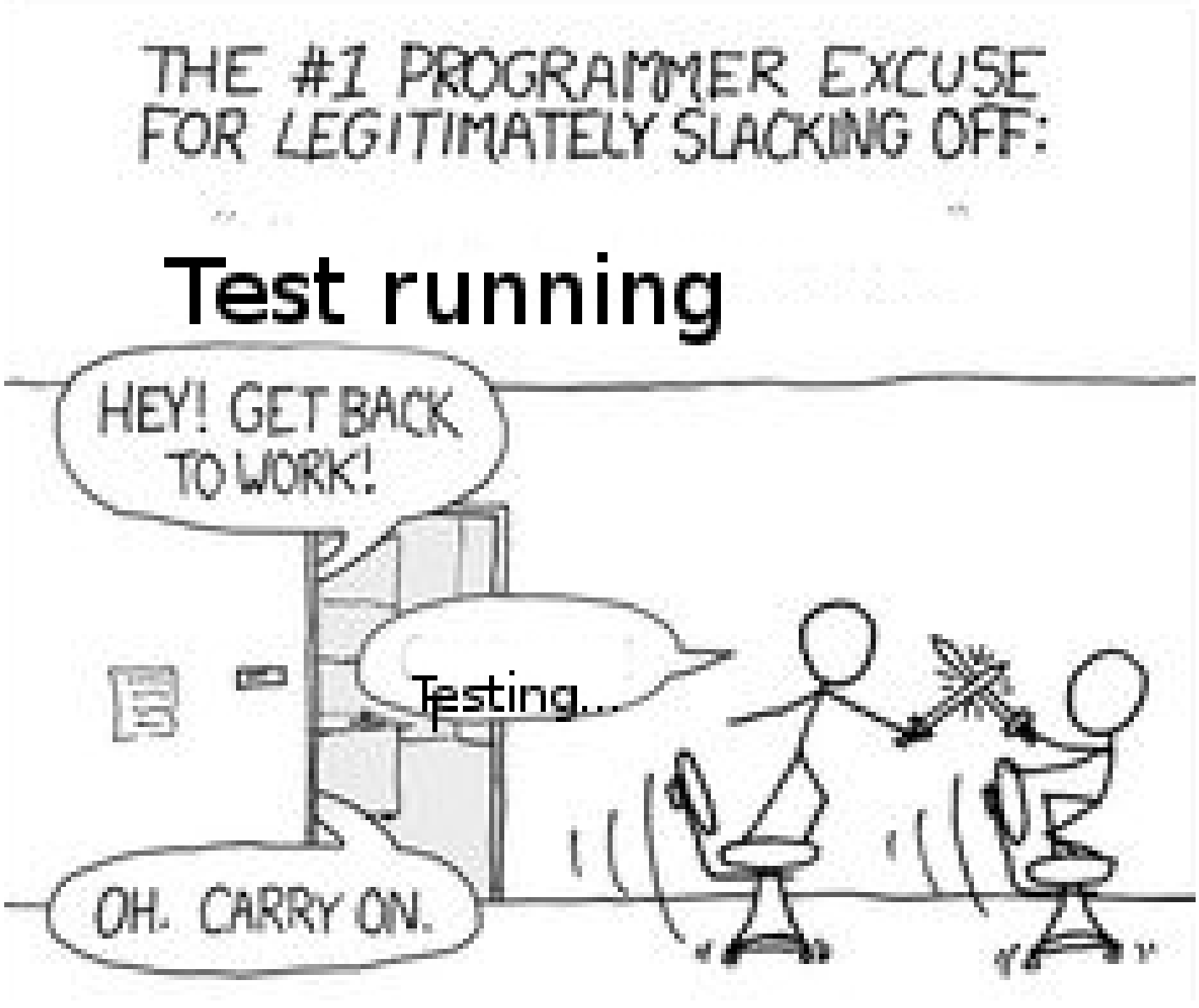
Causes

- *Interacting Tests*
- *Interacting Test Suites*
- *Lonely Test*
- *Resource Leakage*
- *Resource Optimism*
- *Unrepeatable Test*
- *Test Run War*
- *Nondeterministic Test*

Source: `http://xunitpatterns.com/Erratic%20Test.html`

Slow Tests

The tests take too long to run.



Slow Tests

The tests take too long to run.

Symptoms

The tests take long enough to run that developers don't run them every time they make a change to the system under test (SUT). They wait until the next coffee break or other interrupt before running them. Or, whenever they run the tests they walk around and chat with other team members (or play Doom or surf the Internet or . . .)

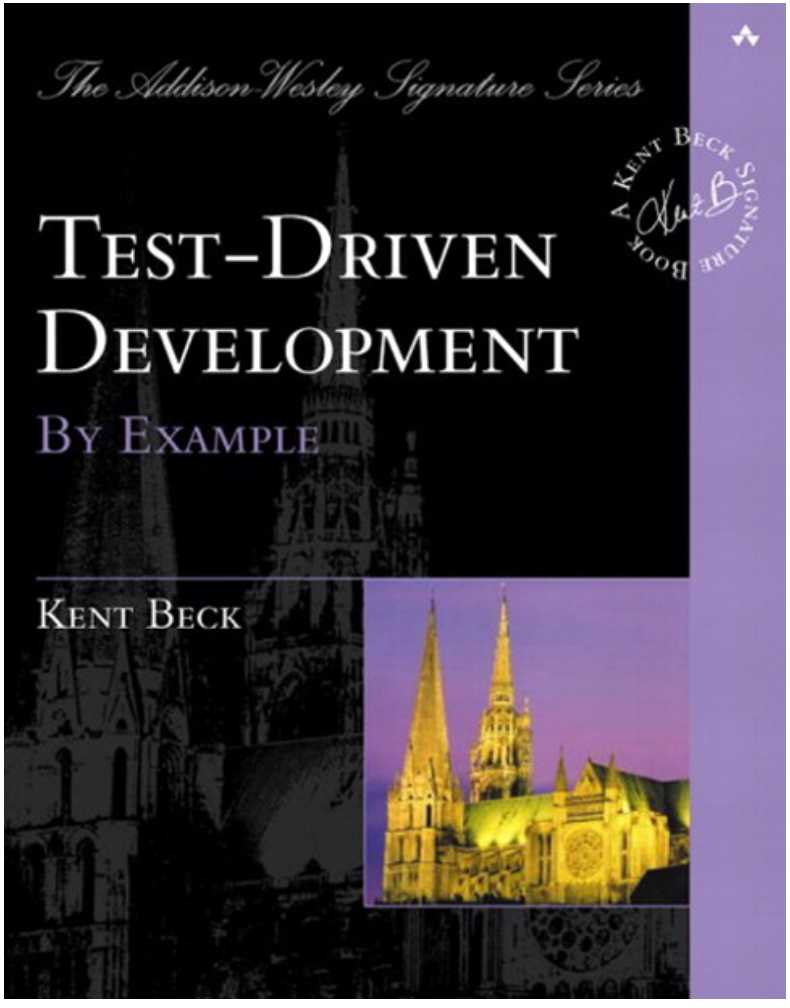
Causes

- *Slow Component Usage*
- *General Fixture*
- *Asynchronous Test*
- *Too Many Tests*

Source: `http://xunitpatterns.com/Slow%20Tests.html`

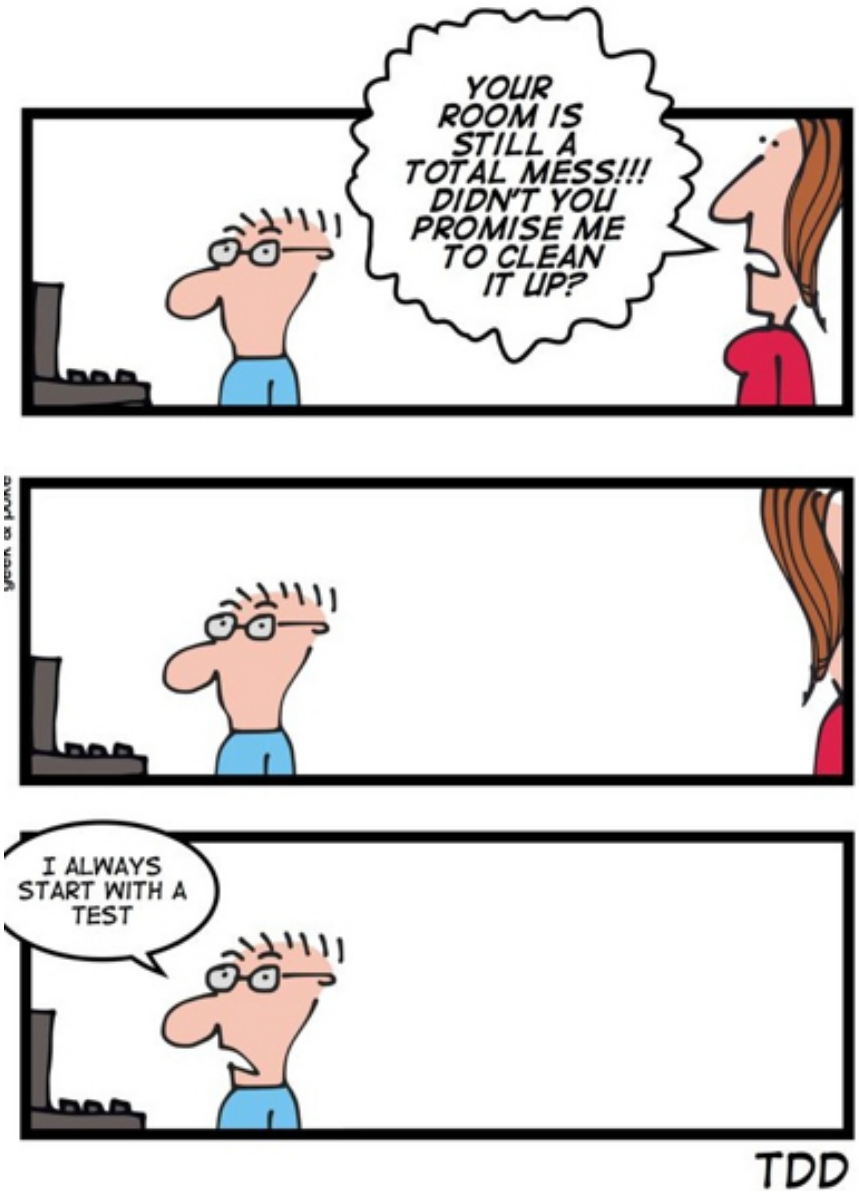
7. L’approche TDD

TDD



= Développement piloté par les tests

Qu’est-ce que le TDD ?



Source: `http://www.datamation.com/imagesvr_ce/2411/tdd.jpg`

Qu’est-ce que le TDD ?

- TDD = *Test Driven Development*

- Origine = une règle de base de XP =

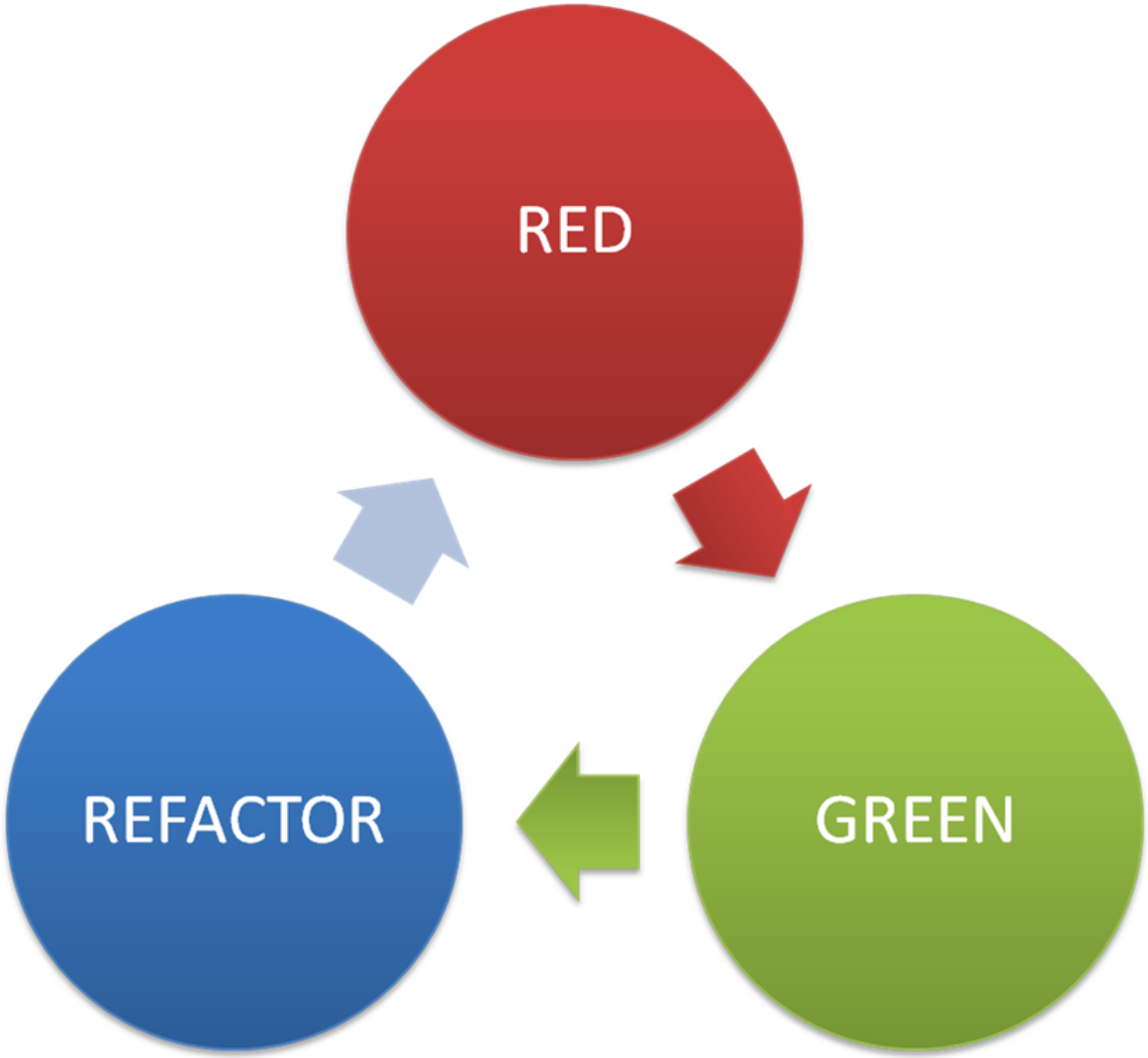
Qu’est-ce que le TDD ?

- TDD = *Test Driven Development*

- Origine = une règle de base de XP =

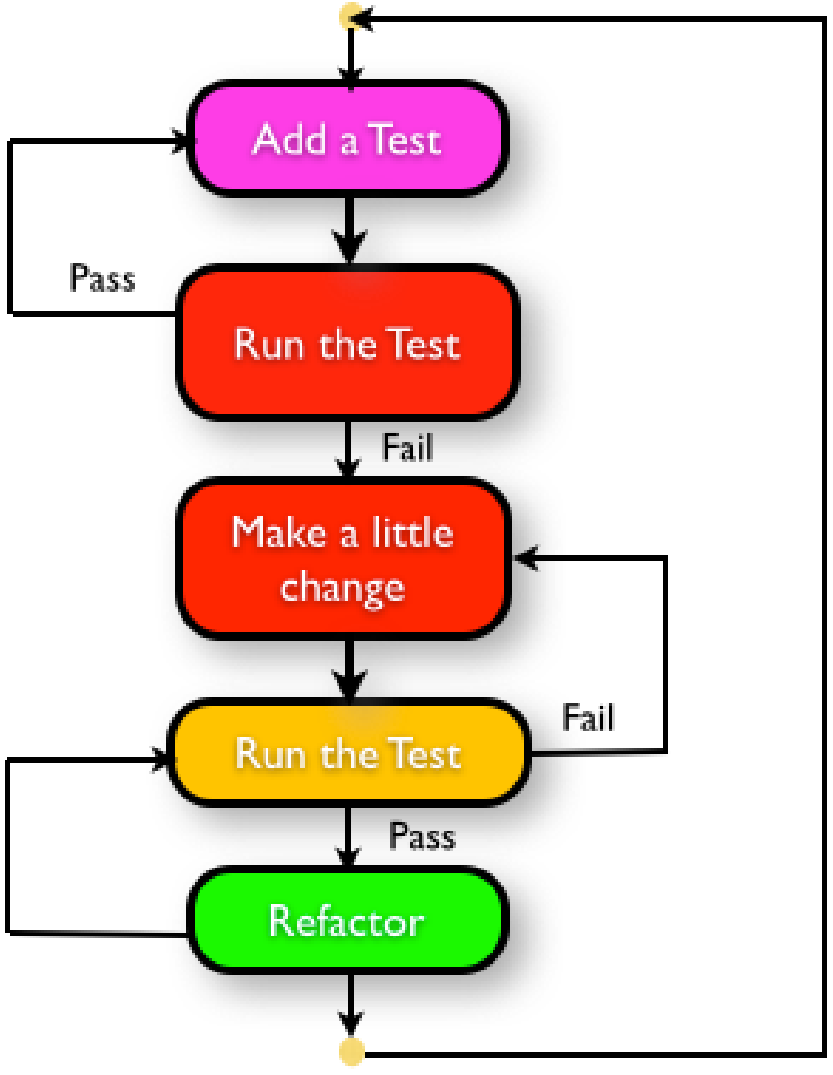
«Code the unit test first !»

L’approche TDD «pure et dure»



Source: `http://manojjaggavarapu.files.wordpress.com/2012/07/redgreenrefacor.png`

Approche TDD «pure et dure» (bis)



Source: `http://agilefaqs.com/services/training/test-driven-development`

Les avantages d'écrire les tests avant le code

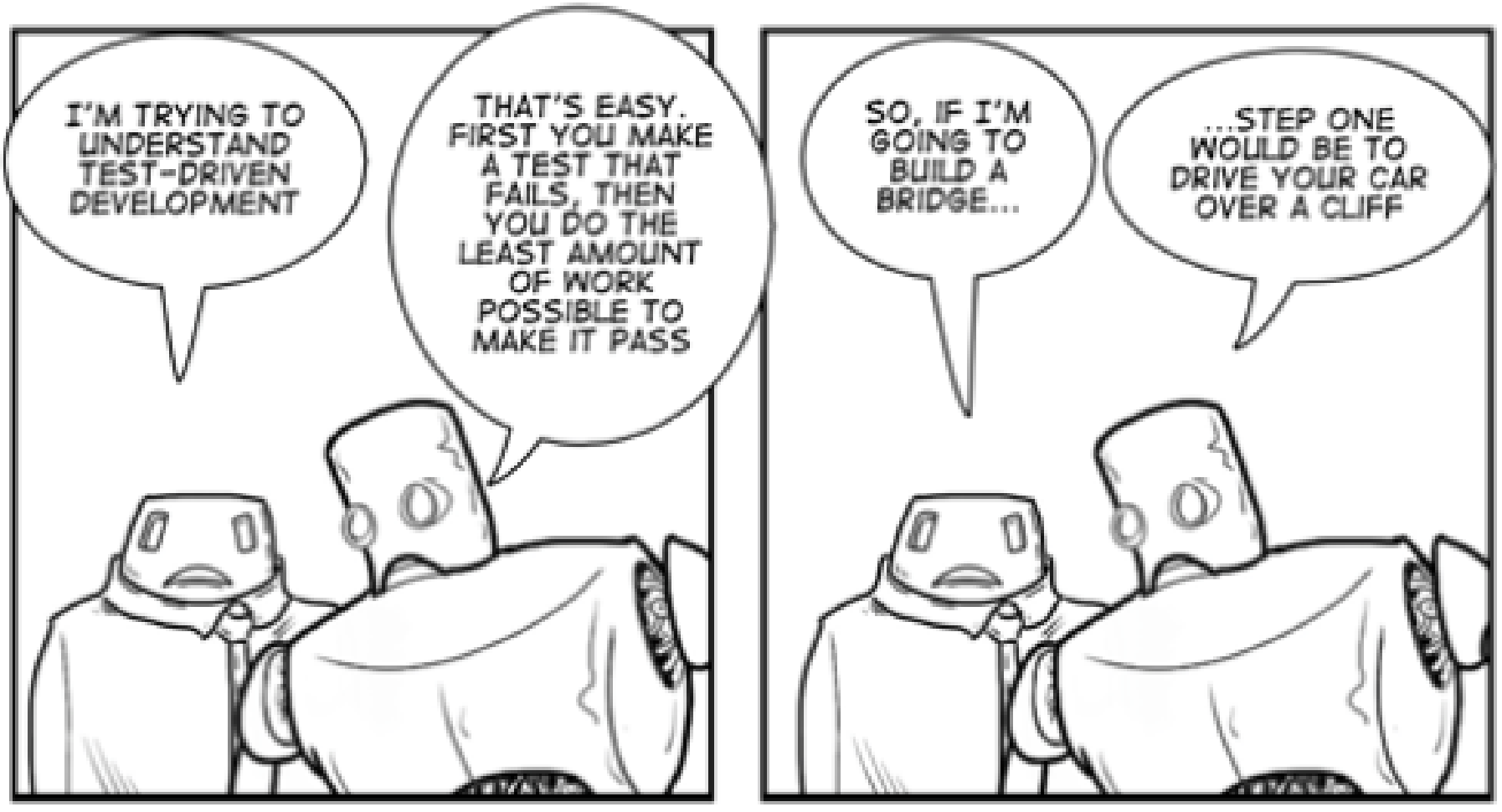
- On est certain d'avoir des tests !
- On est certain que le code est testable
- Les tests utilisent le code, donc aident à définir l'API

Les avantages d'écrire les tests avant le code

- On est certain d'avoir des tests !
- On est certain que le code est testable
- Les tests utilisent le code, donc aident à définir l'API

= *Test Driven Design*

Un aspect controversé de TDD



Un aspect controversé de TDD

Extrait vidéo d'«*Uncle Bob*» (Robert C. Martin) :
`https://www.youtube.com/watch?v=KtHQGs3zFAM`

[0m00s à 2m25s]

Les trois lois de TDD selon *Uncle Bob*

1. *You are not allowed to write any production code unless it is to make a failing unit test pass.*

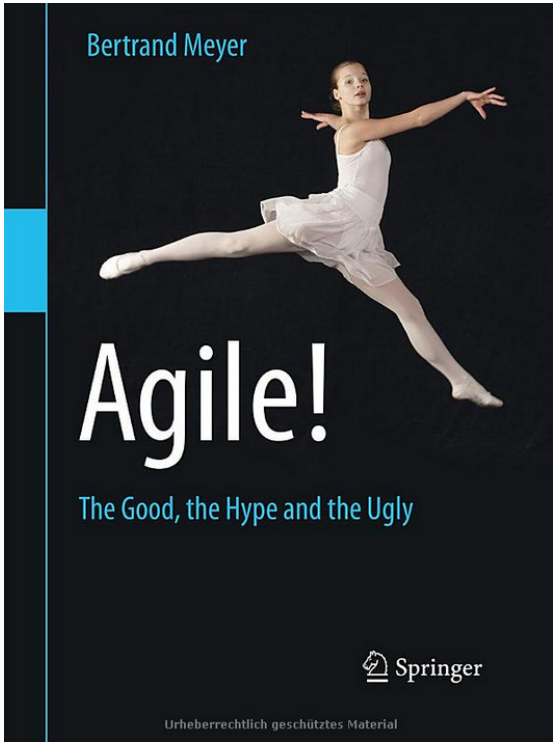
2. *You are not allowed to write any more of a unit test than is sufficient to fail ; and compilation failures are failures.*

3. *You are not allowed to write any more production code than is sufficient to pass the one failing unit test.*

Approche *test-first* «pragmatique» (Meyer, 2014)

«In practice, few organizations apply the strict TDD process in the form of the repetition of the sequence of steps described above.

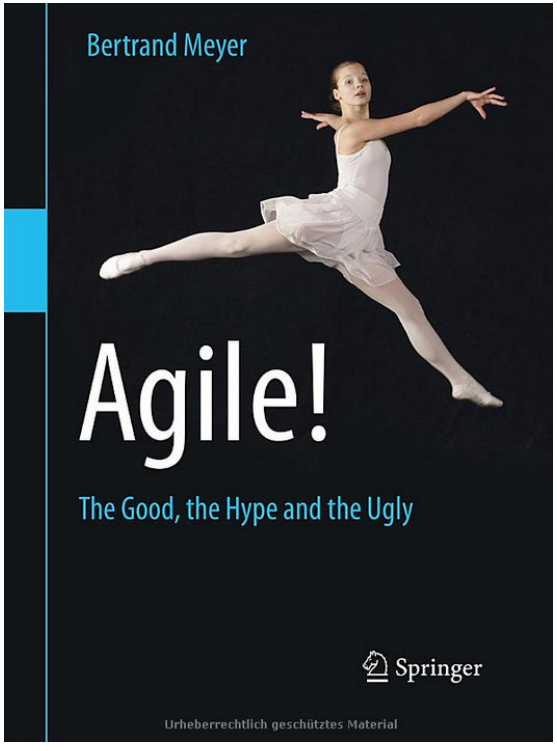
The real insight [is] the idea that any new code must be accompanied by new tests. It is not even critical that the code should come only after the test [. . .] : **what counts is that you never produce one without the other**.»



Approche *test-first* «pragmatique» (Meyer, 2014)

«In practice, few organizations apply the strict TDD process in the form of the repetition of the sequence of steps described above.

The real insight [is] the idea that any new code must be accompanied by new tests. It is not even critical that the code should come only after the test [. . .] : **what counts is that you never produce one without the other**.»



Self-testing code !

Approche *test-first* «pragmatique» (Beust, 2008)

«When it comes to testing, I live by the following rules of thumb :

- "Tests first" or "tests last" is unimportant as long as there are tests.
- Try to think about testing as early as possible in your development process.
- Don't let one liners contradict your experience. For example, don't listen to people who tell you to write "the simplest possible thing that could possibly work", also known as YAGNI. If your experience tells you you're going to need this extra class in the future even if it's not needed right now, follow your judgment and add it now.
- Keep in mind that functional tests are the only tests that really matter to your users. Unit tests are just a convenience for you, the developer. A luxury. If you have time to write unit tests, great : they will save you time down the road when you need to track bugs. But if you don't, make sure that your functional tests cover what your users expect from your product.»

Source: `http://beust.com/weblog/2008/03/03/tdd-leads-to-an-architectural-meltdown-around-iteration-three/`

Un avantage d'avoir des tests et du *self-testing code*

Tests eliminate fear

Robert C. Martin

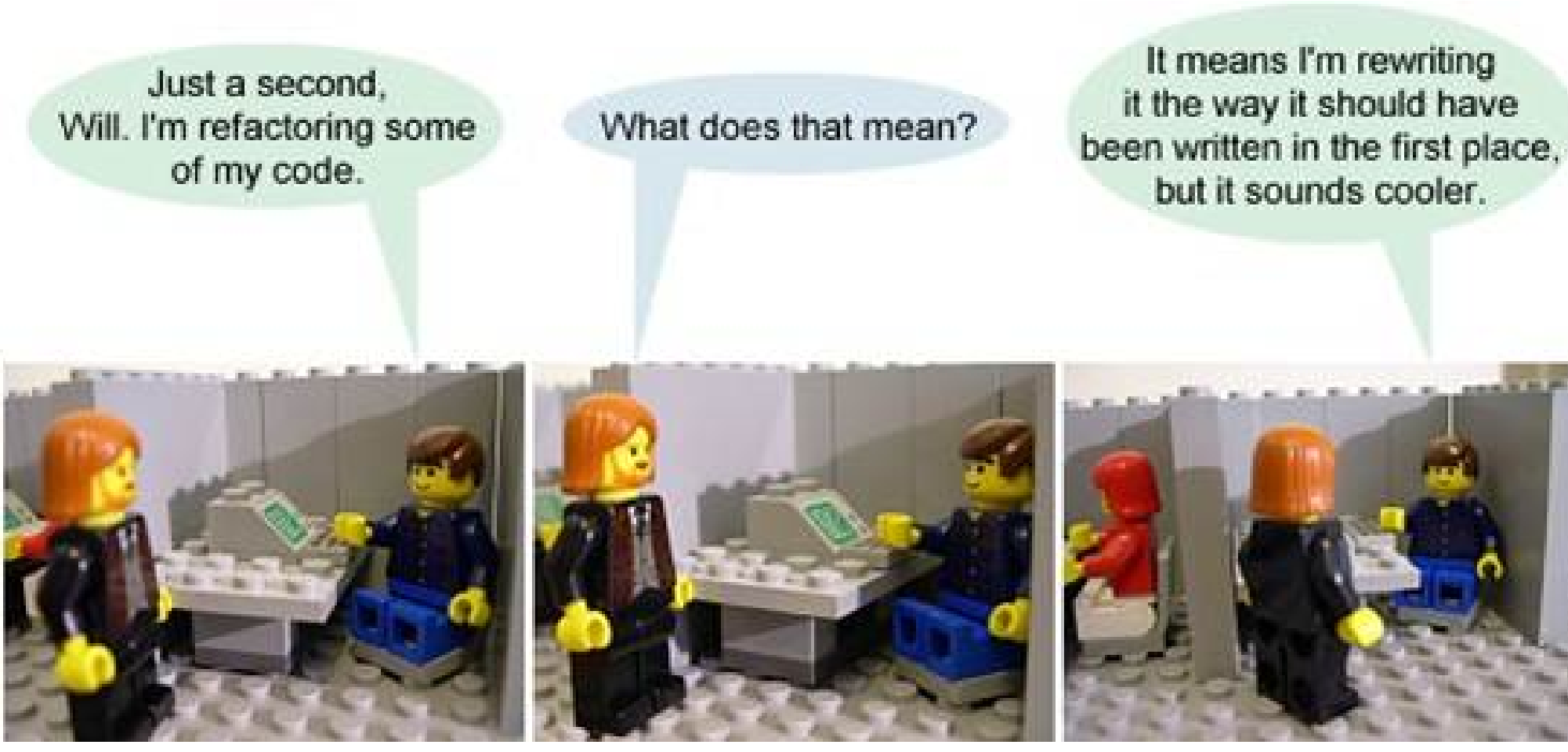
Un avantage d'avoir des tests et du *self-testing code*

Tests eliminate fear

Tests allow you to make changes without the risk of breaking something

Robert C. Martin

La présence de tests permet/favorise le *refactoring*



Source: `http://web.cse.ohio-state.edu/~crawfis/CSE3902/RefactoringCartoon.jpg`

Effets de l'essor de TDD — *hard* ou *light*

- De nombreux cadres de tests sont disponibles (152) :
  `http://c2.com/cgi/wiki?TestingFramework`
- On est plus conscients de l'importance des tests
- D'autres outils liés aux tests se sont développés, par ex., en Ruby :
  - autotest (voir prochaine diapositive)
  - simplecov
  - ...
- Les tests sont devenus plus simples à spécifier et exécuter

$\Rightarrow$ Comment bien définir des tests ?

D’autres outils liés aux tests : `autotest`

`autotest`

- Relance l’exécution des tests aussitôt qu’on modifie un fichier — aussitôt qu’on sauvegarde des modifications avec l’éditeur de texte.
- Si on respecte certaines conventions, ne va lancer que les tests des éléments (code applicatif ou code de tests) ayant été modifiés.

Il est important d'avoir une discipline de travail qui est «professionnelle»

Extrait vidéo d'«*Uncle Bob*» (Robert C. Martin) :
`https://www.youtube.com/watch?v=YX3iRjKj7C0`

[53m15s à 54m42s]
[59m00s à 1h00m40s]

Références

K. Beck.

Test-Driven Development—By Example.
Addison-Wesley, 2003.

D. Chelimsky, D. Astels, Z. Dennis, A. Hellesoy, B. Helmkamp, and D. North.

The RSpec Book : Behaviour Driven Development with RSpec, Cucumber, and Friends.
The Pragmatic Bookshelf, 2010.

J. Fields.

Working Effectively with Unit Tests.
LeanPub, 2014.

A. Hunt and D. Thomas.

The Pragmatic Programmer—From Journeyman to Master.
Addison-Wesley, 2000.

G. Meszaros.

xUnit Test Patterns—Refactoring Test Code.
Addison-Wesley, 2007.

B. Meyer.

Agile ! : The Good, the Hype and the Ugly.
Springer, 2014.

V. Subramaniam and A. Hunt.

Practices of an Agile Developer—Working in the Real World.
The Pragmatic Bookshelf, 2006.

Références

«*Writing a Test Framework from Scratch*», par Ryan Davis (RailsConf 2016) :

- Une vidéo ($\approx$ une demi-heure) qui présente le développement, à partir de zéro (*from scratch*), d'un cadre de tests simple.
- `https://www.youtube.com/watch?v=VPr5pmlAq20`

Ce cadre de tests, appelé `microtest`, est une version simplifiée de `MiniTest`.[1]

1. Davis est le concepteur de `MiniTest`.

A. Une version simplifiée d'une méthode `it` de style `MiniTest`

Définition Ruby d'une méthode `it` (simplifiée)

Définition d'une méthode `symbole_pour`

```
def symbole_pour( prefixe, chaine )
  (prefixe + chaine.gsub(/\s/, "_")).to_sym
end
```

Exemples d'appels de la méthode `symbole_pour`

```
=> :symbole_pour
>> symbole_pour "foo_", "abc"
=> :foo_abc

>> symbole_pour "test_", "de methode avec un long nom"
=> :test_de_methode_avec_un_long_nom
```

Définition Ruby d'une méthode `it` (simplifiée)

Méthode `it`

```ruby
def it( nom_du_test, &bloc )
  define_method symbole_pour("test_", nom_du_test) do
    instance_eval &bloc
  end
end
```