Tests unitaires et tests d'acceptation

Guy Tremblay
Professeur

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

INF600A
2 octobre 2018
6 novembre 2018 (?)

1. Introduction/motivation
2. Les différents niveaux de tests
3. Les tests unitaires
4. Un cadre de tests unitaires pour Ruby : `MiniTest`
   - Approche avec assertions
   - Approche avec *expectations*
5. Le principe d'inversion des dépendances et l'injection de dépendances
6. Les doublures de tests avec `MiniTest`
   - Les *stubs*
   - Les objets *mocks*
   - Autres formes de doublures de tests
7. Les tests unitaires et l'approche TDD
8. Les tests d'acceptation et l'approche BDD
   - Comment peut-on spécifier des tests d'acceptation ?
   - Exécution automatique de tests d'acceptation décrits pas des scénarios utilisateurs : L'outil `cucumber`

A. Un aperçu des tests d'acceptation du devoir 1

1. Introduction/motivation

Les premières choses à faire, quand on veut développer 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.

Les premières choses à faire, quand on veut développer 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.

Les premières choses à faire, quand on veut développer 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 :*

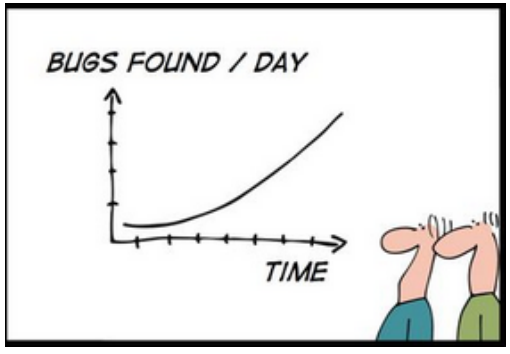
- *Version control*
- **Unit Testing**
- *Build automation*

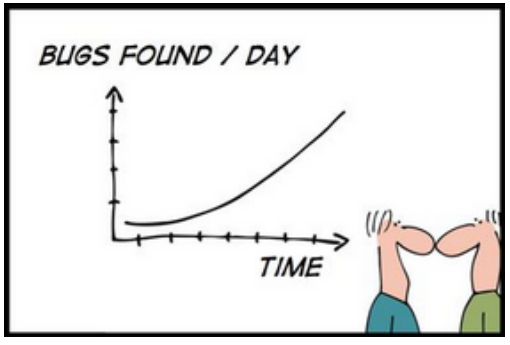
Version control needs to come before anything else. It's the first bit of infrastructure we set up on any project.»

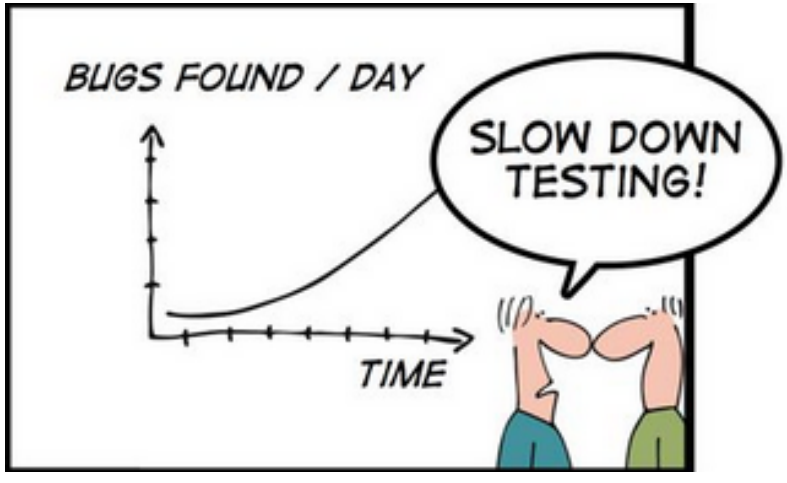
«*Practices of an Agile Developer—Working in the Real World*», Subramaniam & Hunt, 2006.

Pourquoi les tests sont-ils importants ?

Les tests, développés et exécutés tôt, sont cruciaux pour trouver rapidement les bogues







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

Quelques *tips* de Hunt & Thomas sur les tests

Tirés de « *The Pragmatic Programmer—From Journeyman to Master* »

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.

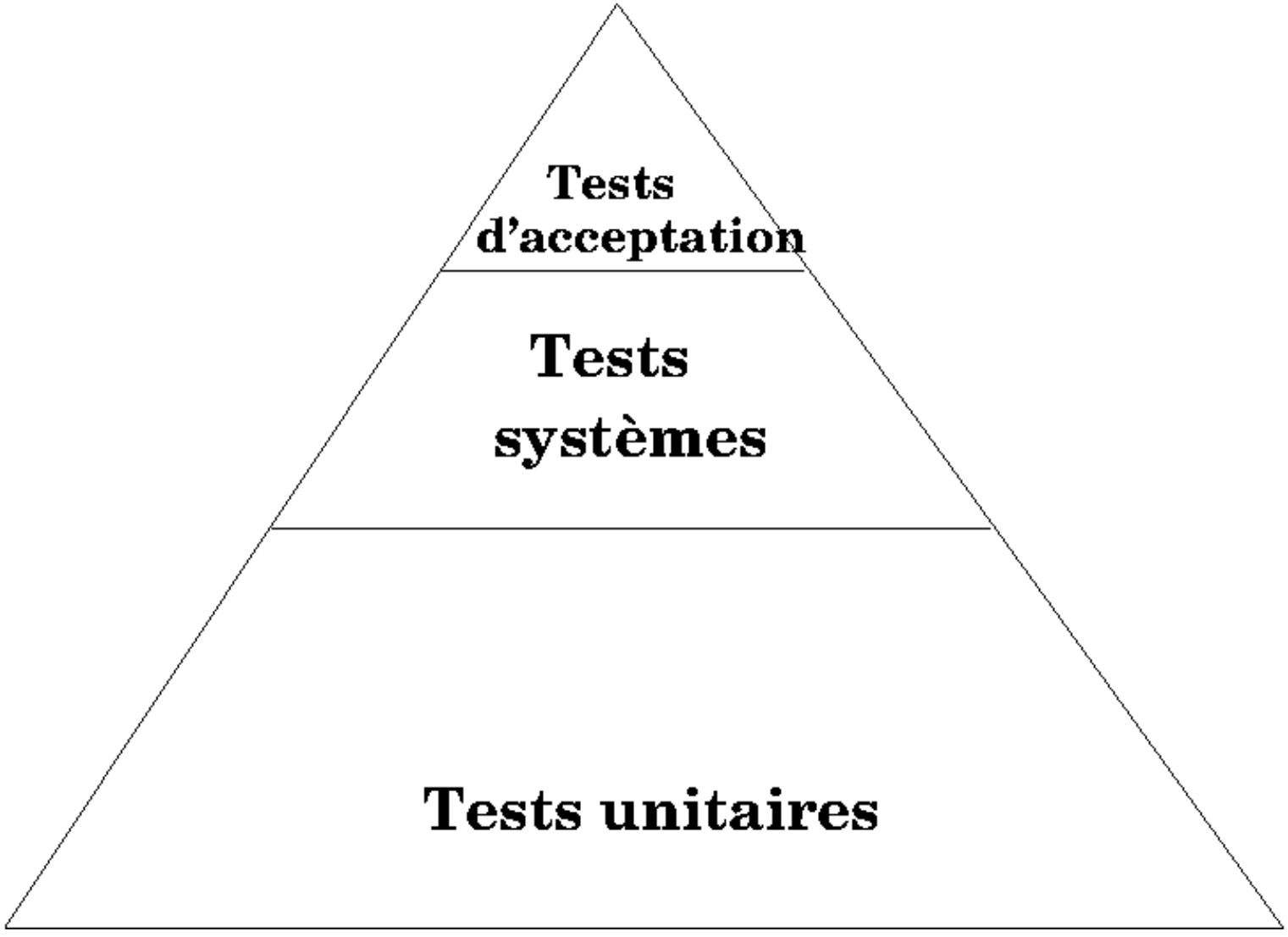
62. Test Early. Test Often. Test Automatically.
    Tests that run with every build are much more effective than test plans that sit on a shelf.

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

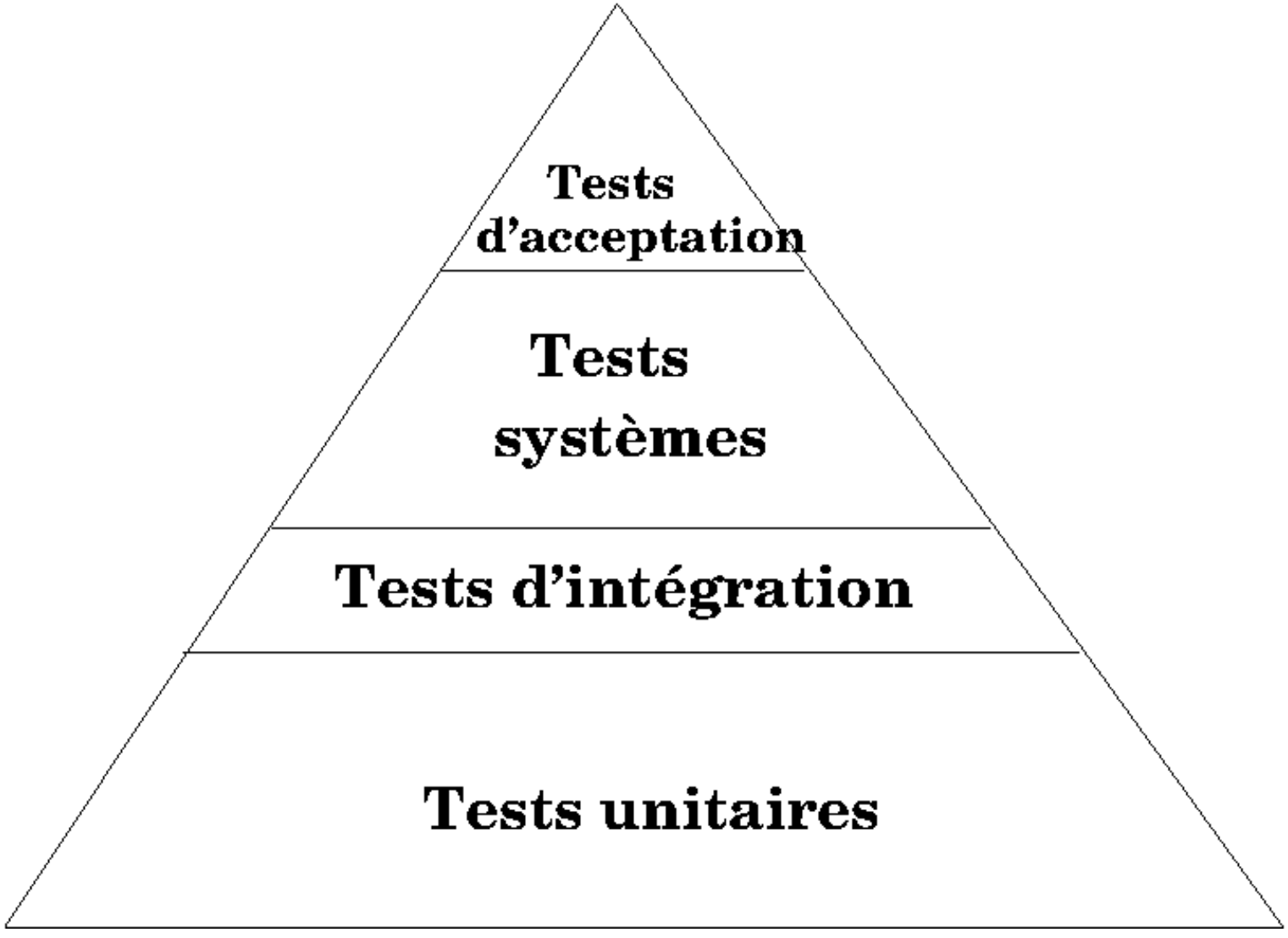
66. Find Bugs Once
    Once a human tester finds a bug, it should be the last time a human tester finds that bug. Automatic tests should check it from then on.

2. Les différents niveaux de tests

Les différents niveaux de tests



Les différents niveaux de tests



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

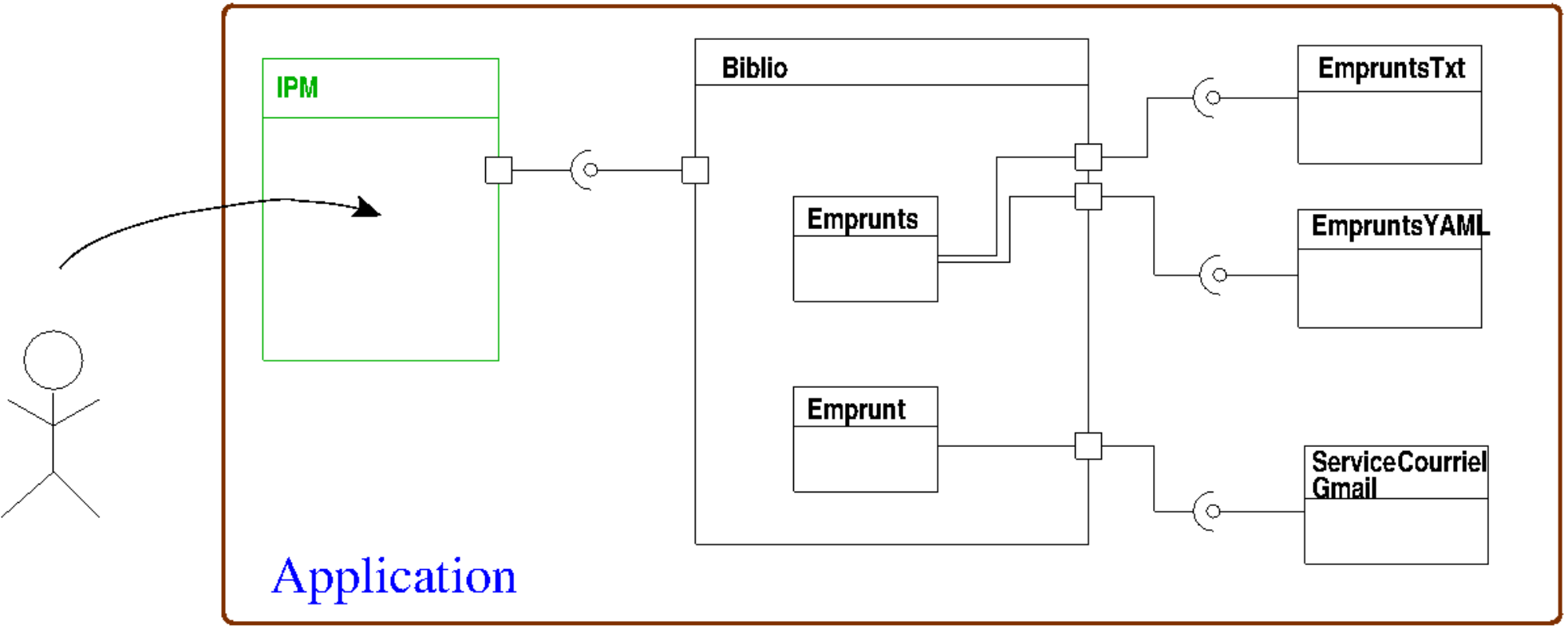
⇒ Conçus sans connaître le code du composant

⇒ Écrits tôt et indépendamment des développeurs

- Typique : Tests de système et d'acceptation

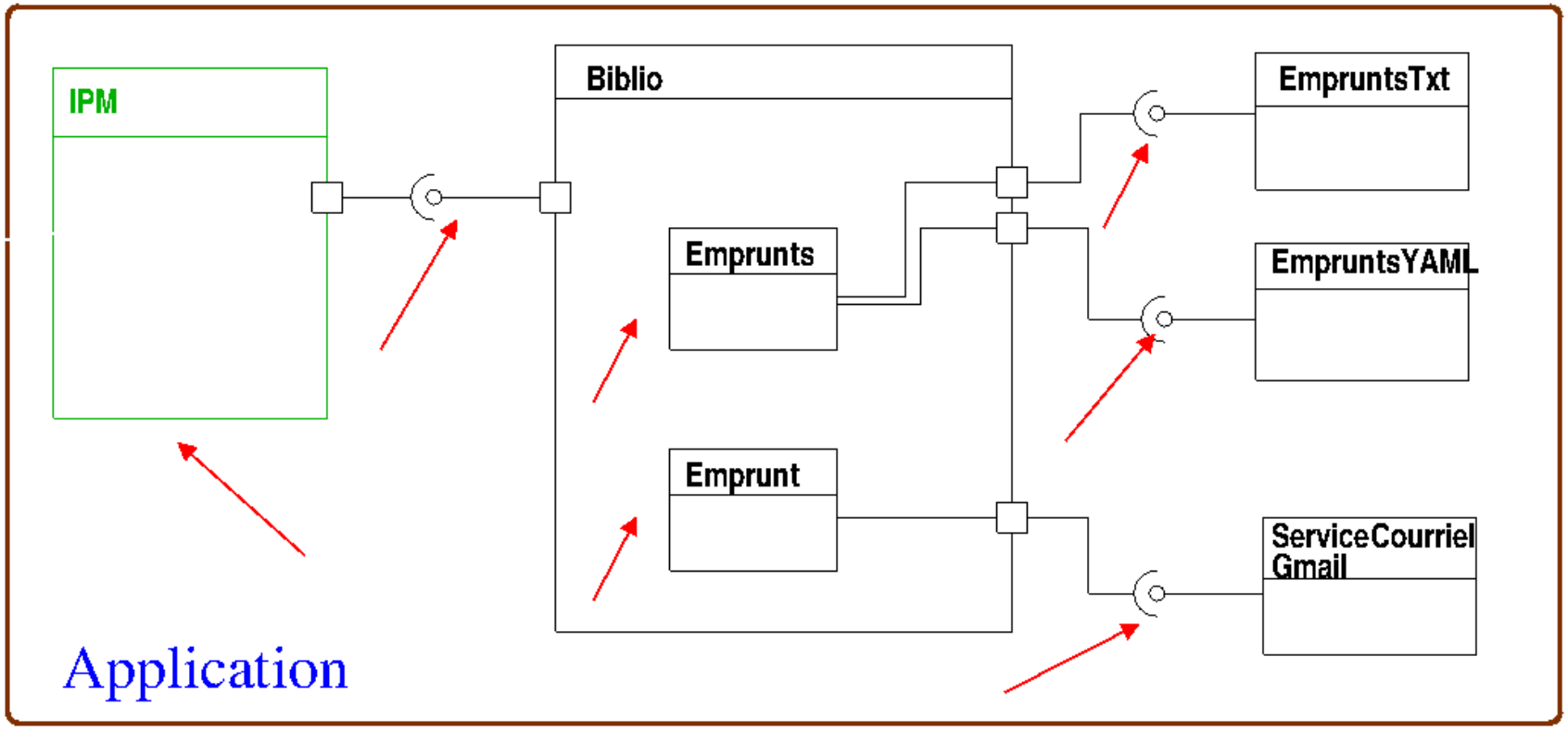
Les différents niveaux de tests

Soit une application composée de plusieurs modules et composants



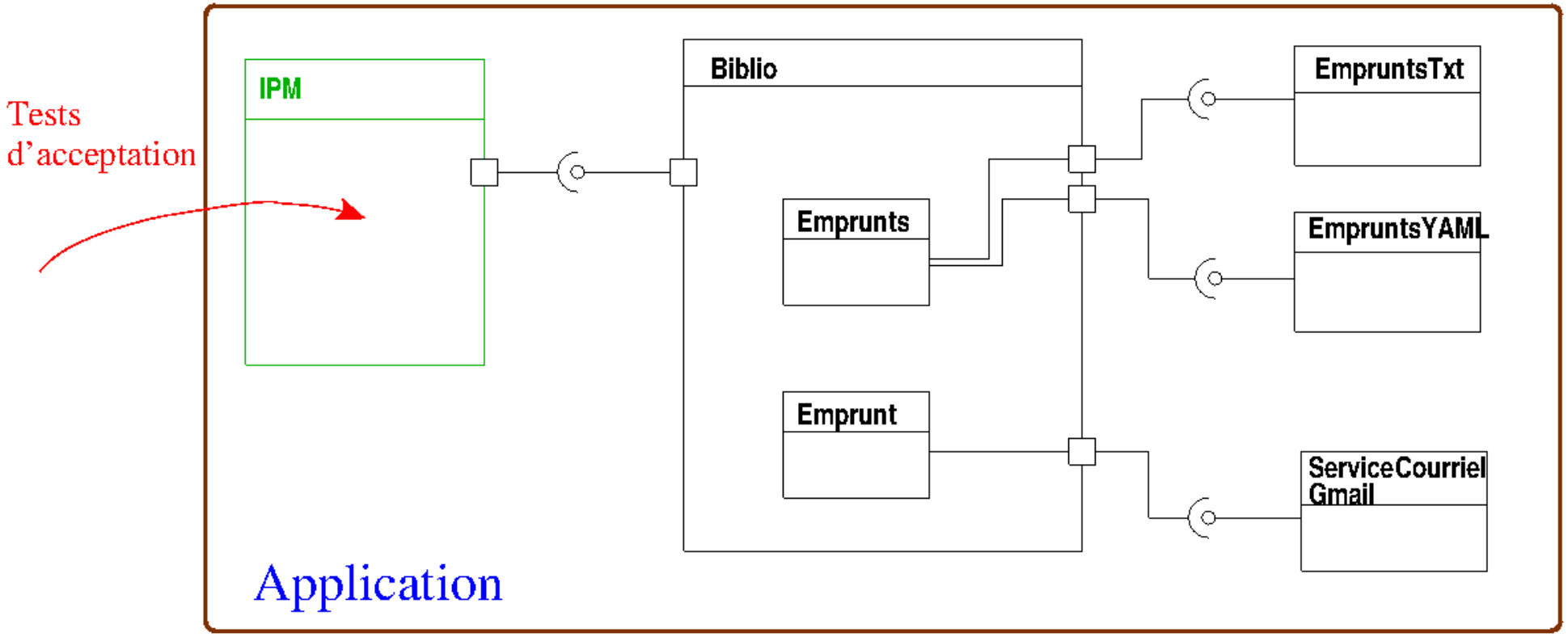
Les différents niveaux de tests

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 !

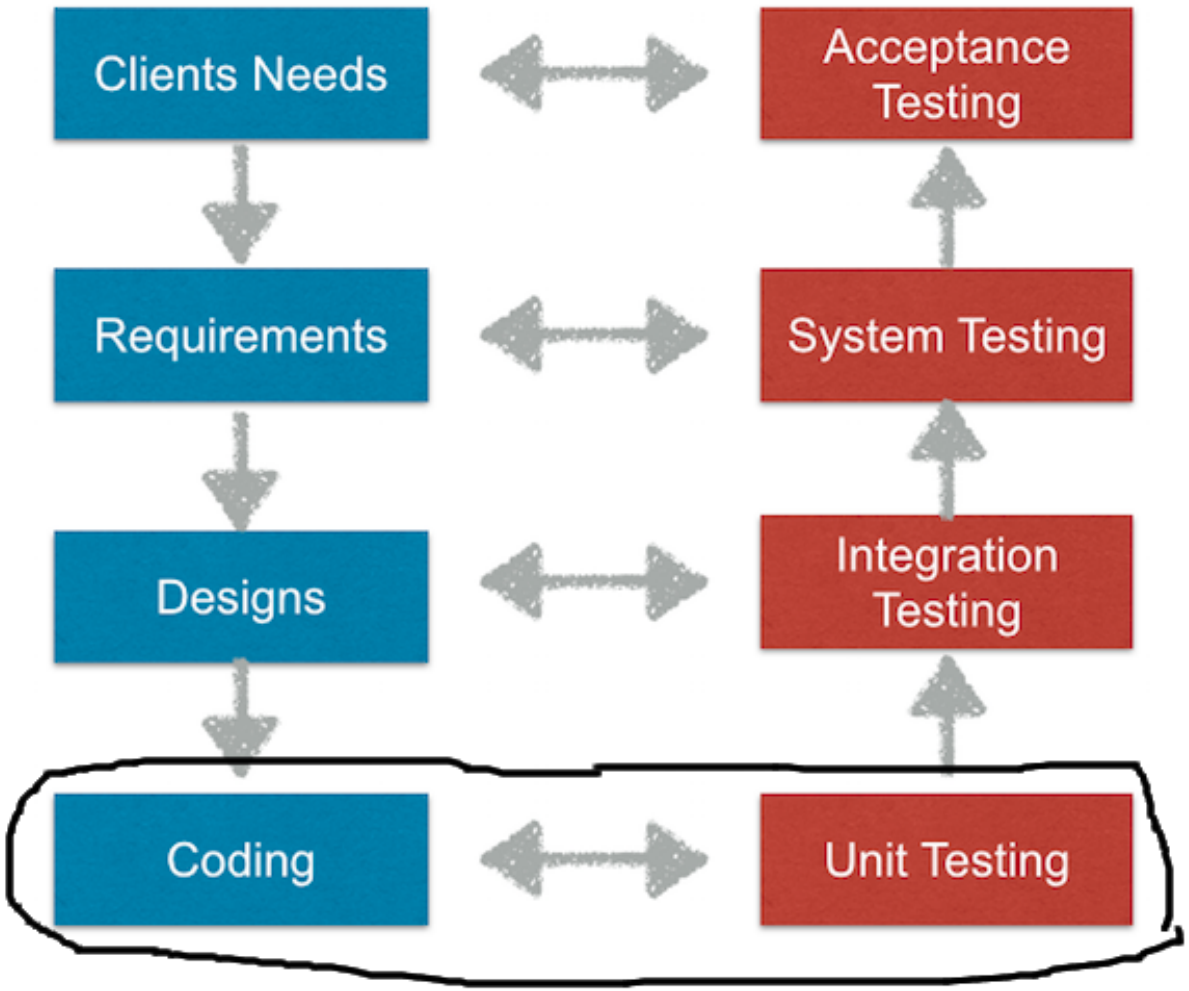
Tant les tests de système que les tests d'acceptation sont des «*end-to-end tests*»

Tests complets, «de bout en bout» = *End-to-end tests*

End-to-end testing involves ensuring that that integrated components of an application function as expected. The entire application is tested in a real-world scenario such as communicating with the database, network, hardware and other applications.

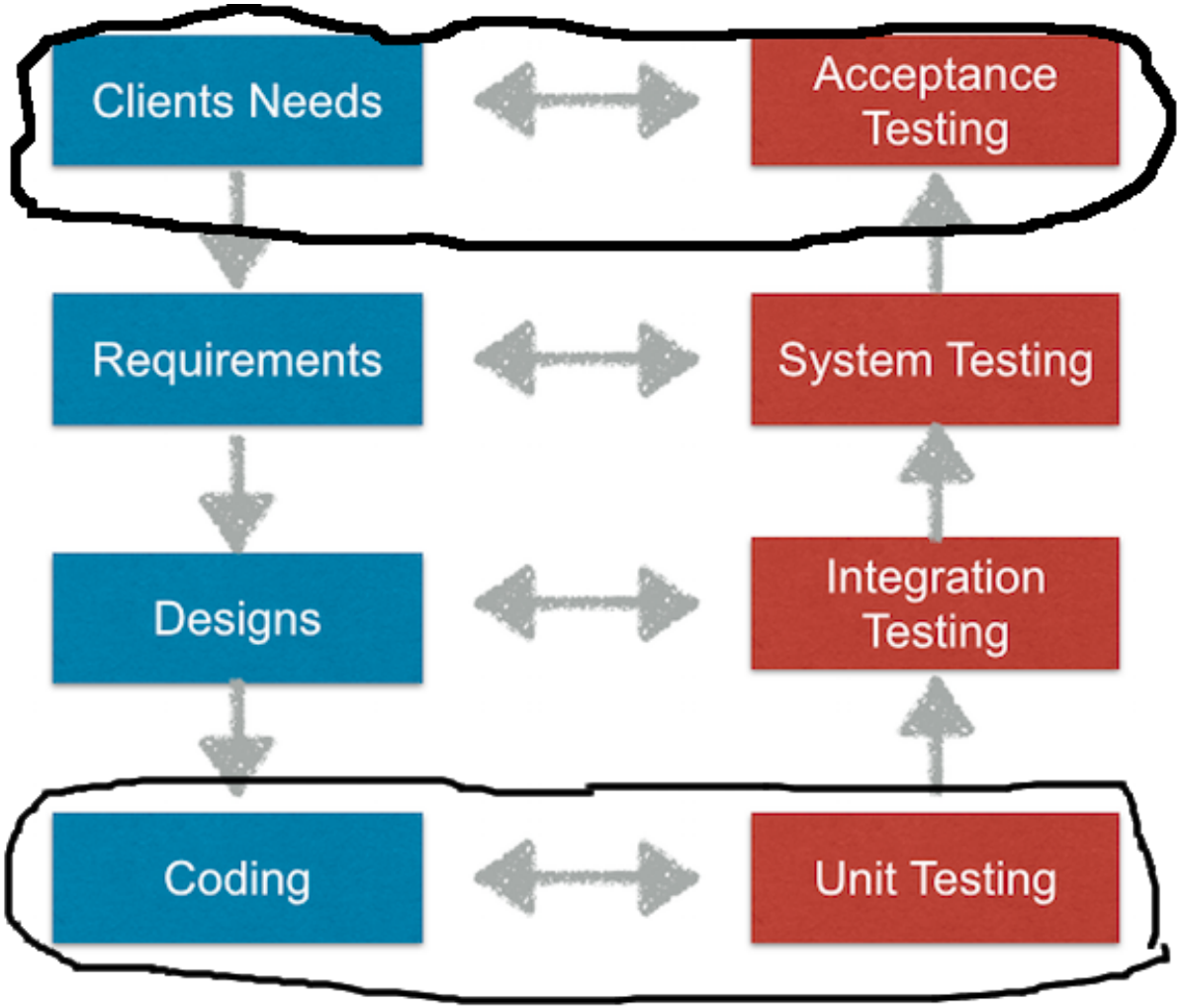
Source: `https://www.techopedia.com/definition/7035/end-to-end-test`

Les différents niveaux de tests



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

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

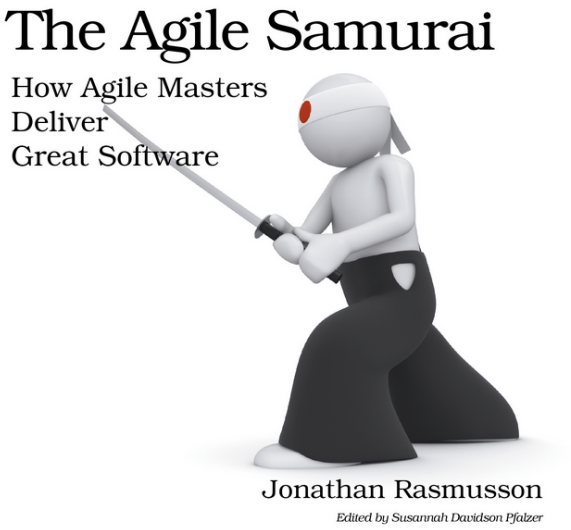
Vérification du fonctionnement du système dans son ensemble.

Tests d'acceptation

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

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.

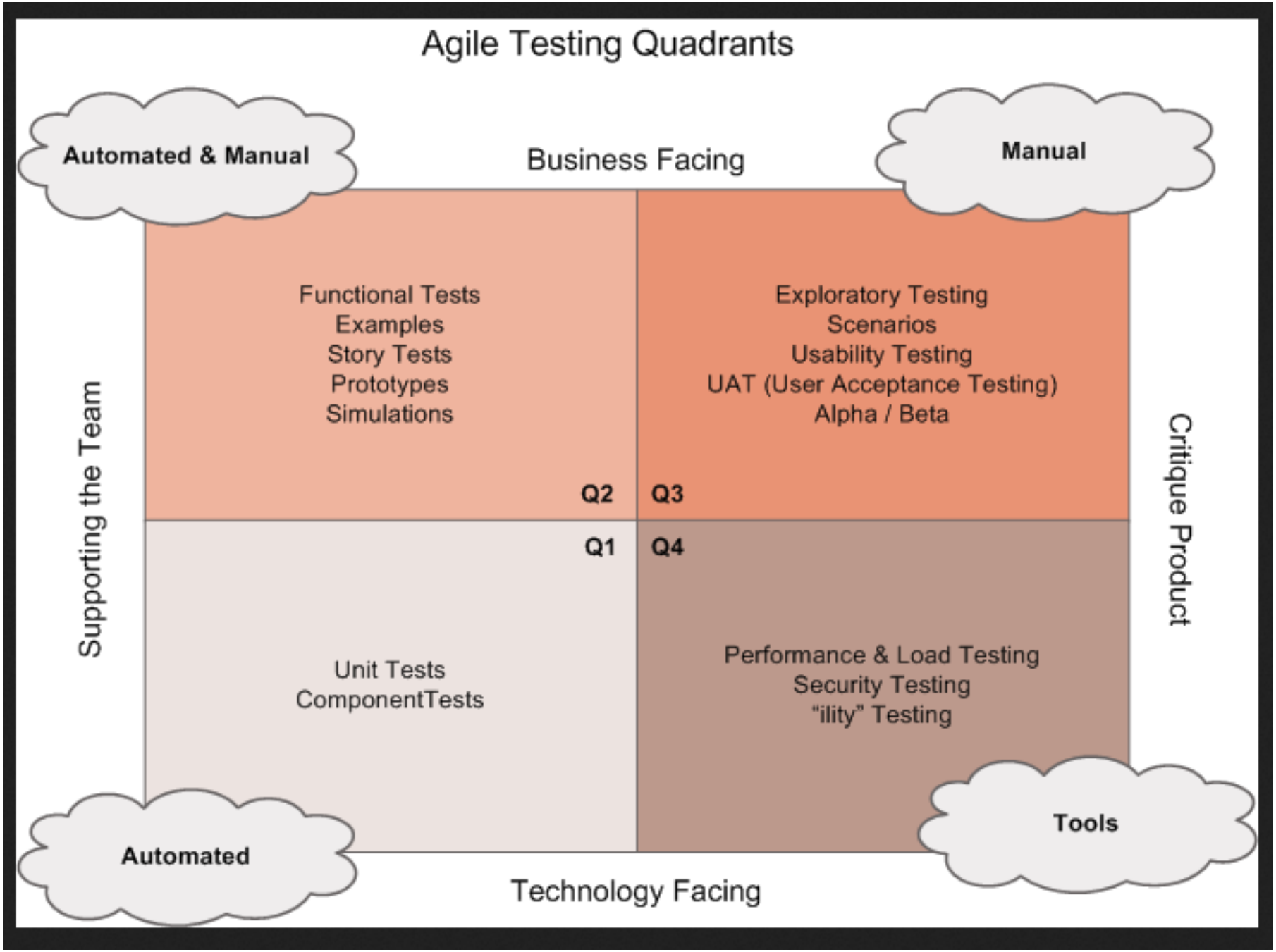


Une définition des tests d'acceptation

Acceptance Test

The complete application (or system) is tested by end users (or representatives) for the purpose of determining readiness for deployment.

Source*: «*The Rational Unified Process—An Introducion (Second Edition)*», Kruchten*



Source: http://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants/

Autre définition des tests d'acceptation

Acceptance tests are tests that define the business value each story must deliver. They may verify functional requirements or nonfunctional requirements such as performance or reliability. Although they are used to help guide development, it is at a higher level than the unit-level tests used for code design in test-driven development. Acceptance test is a broad term that my include both business-facing and technology-facing tests.

Source*:* «Agile Testing—A Practical Guide for Testers and Agile Teams*», Crispin & Gregory*

Autre définition des tests d'acceptation

Acceptance tests are created from user stories. During an iteration the user stories selected during the iteration planning meeting will be translated into acceptance tests. The customer specifies scenarios to test when a user story has been correctly implemented. A story can have one or many acceptance tests, what ever it takes to ensure the functionality works.

Acceptance tests are black box system tests. Each acceptance test represents some expected result from the system.

The name acceptance tests was changed from functional tests. This better reflects the intent, which is to guarantee that a customers requirements have been met and the system is acceptable.

Source: `http://www.extremeprogramming.org/rules/functionaltests.html`

Les tests 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 peuvent nécessiter un grand nombre de lignes de code

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		

Tests fonctionnels :

Fichiers	**KLOC**

Les tests peuvent nécessiter un grand nombre de lignes de code

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**

Les tests peuvent nécessiter un grand nombre de lignes de code

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

Les tests peuvent nécessiter un grand nombre de lignes de code

SQLite

- Version 3.23.0 (2018-04-02)
- Bibliothèque écrite en C
- SLOC = *Source Lines Of Code*

  (sans compter lignes blanches et commentaires)

Partie du Code SQLite	**KSLOC**
Code fonct.	128.9
Tests	?

Source : «*How SQLite Is Tested*»

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

Les tests peuvent nécessiter un grand nombre de lignes de code

SQLite

- Version 3.23.0 (2018-04-02)
- Bibliothèque écrite en C
- SLOC = *Source Lines Of Code*

  (sans compter lignes blanches et commentaires)

Partie du Code SQLite	**KSLOC**
Code fonct.	128.9
Tests	91772.0

Source : «*How SQLite Is Tested*»

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

3. Les tests unitaires

Pourquoi les tests unitaires sont particulièrement 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 fonctionnement du composant avant son intégration avec d'autres composants
- Fournissent des exemples d'utilisation du composant

Pourquoi les tests unitaires sont particulièrement importants (suite)

- Permettent d'obtenir rapidement du feedback

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

- Diminuent les coûts des tests de régression

$\Rightarrow$ Assurent la non régression

- Permettent de faire du *refactoring* avec confiance

Pourquoi les tests unitaires sont particulièrement importants (suite)

- Permettent d'obtenir rapidement du feedback

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

- Diminuent les coûts des tests de régression

$\Rightarrow$ 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`

Pourquoi les tests unitaires sont particulièrement importants (suite)

- Permettent d'obtenir rapidement du feedback

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

- Diminuent les coûts des tests de régression

$\Rightarrow$ 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 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*”.

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

Stratégies d'exécution des tests unitaires

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 asssociation au composant testé.
- Permettent d'exécuter de façon automatique une ou plusieurs suites de tests.
- Reposent sur l'utilisation d'assertions :
  - 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 ou «bidons», etc.

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`

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

Le cadre de test 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

Le cadre de test le plus connu = JUnit

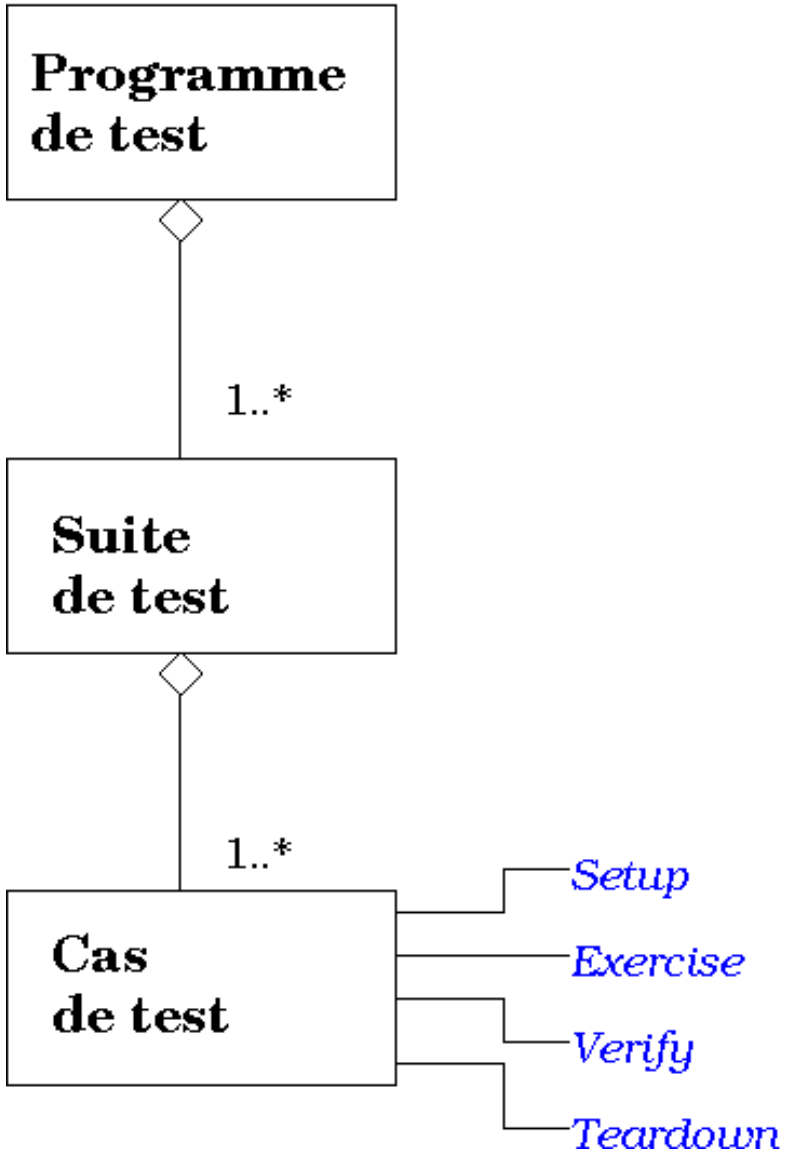
- Cadre de tests pour Java
- Développé et popularisé par les promoteurs de XP (*eXtreme Programming*) — notamment, Kent Beck.
- A rendu «populaire» l'écriture de tests unitaires et l'utilisation de cadres de test

De (très !) nombreux cadres de tests sont disponibles

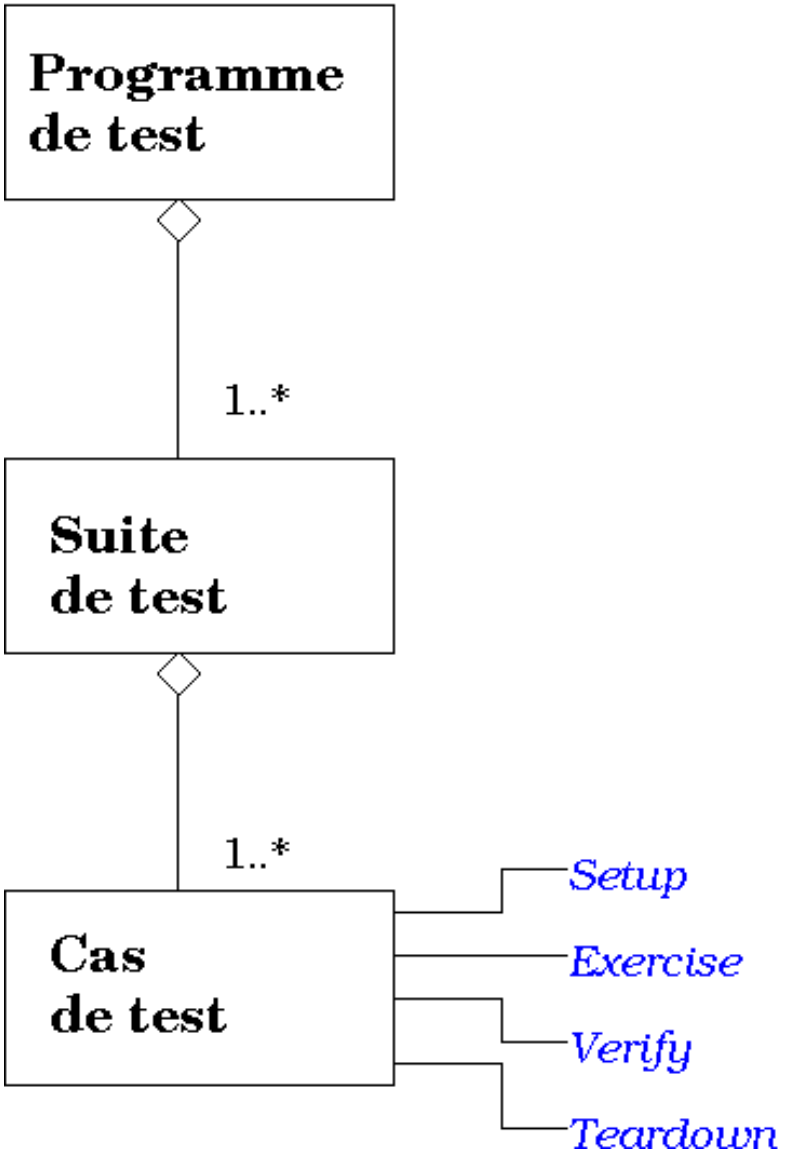
Des cadres équivalents existent pour d'autres langages

- `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 170 : `http://c2.com/cgi/wiki?TestingFramework`
- Par ex., cadres de tests pour le langage C : `GUNit`, `Check`, `MinUnit`, `CUnit`, `CuTest` + 9 autres.

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 produit — ou l'effet obtenu — est celui désiré
- (*Teardown*) On nettoie

4. Un cadre de tests unitaires pour Ruby : `MiniTest`

Pourquoi regarde-t-on `MiniTest` dès maintenant, même si c'est en Ruby ?

- Fait : `MiniTest` est un cadre de tests unitaires

- Fait : Ruby étant un langage de script, on peut facilement l'utiliser comme *glue language* —

Pourquoi regarde-t-on `MiniTest` dès maintenant, même si c'est en Ruby ?

- Fait : `MiniTest` est un cadre de tests unitaires
- Fait : Ruby étant un langage de script, on peut facilement l'utiliser comme *glue language* — pour lancer l'exécution de programmes externes

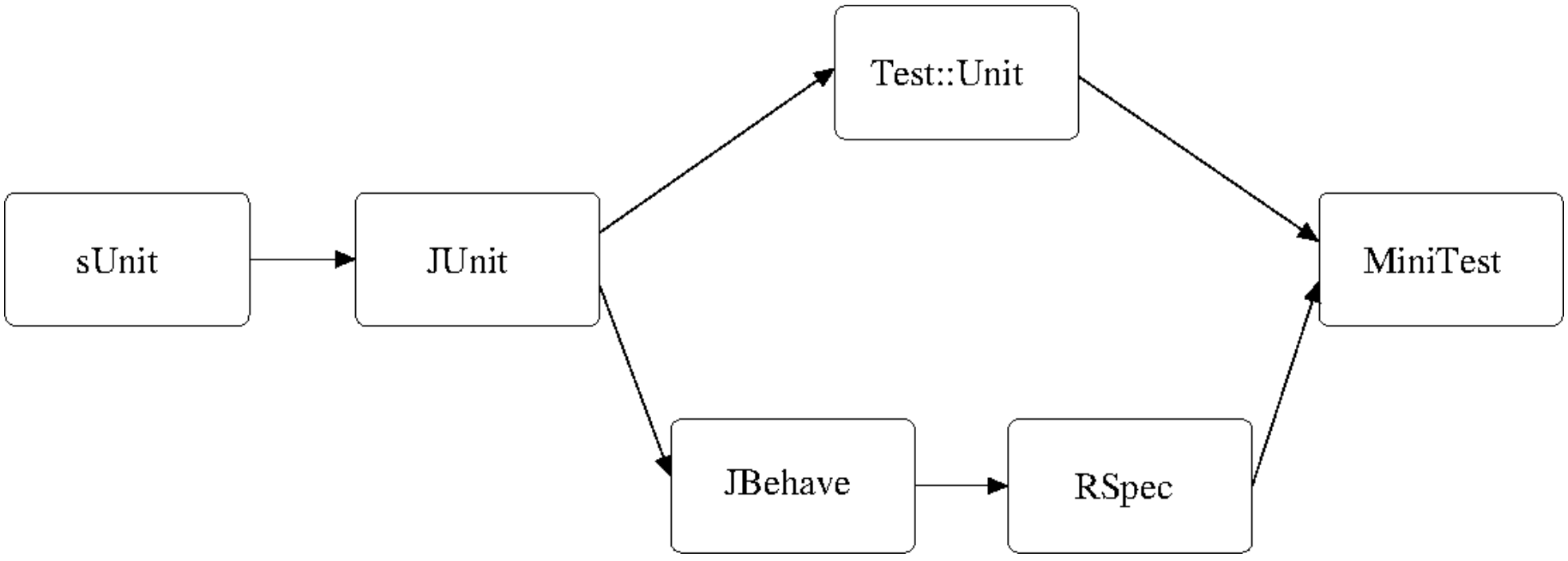
Pourquoi regarde-t-on `MiniTest` dès maintenant, même si c'est en Ruby ?

- Fait : `MiniTest` est un cadre de tests unitaires
- Fait : Ruby étant un langage de script, on peut facilement l'utiliser comme *glue language* — pour lancer l'exécution de programmes externes
- Donc, il est facile d'utiliser Ruby et `MiniTest`... pour faire des tests d'acceptation...

Pourquoi regarde-t-on `MiniTest` dès maintenant, même si c'est en Ruby ?

- Fait : `MiniTest` est un cadre de tests unitaires
- Fait : Ruby étant un langage de script, on peut facilement l'utiliser comme *glue language* — pour lancer l'exécution de programmes externes
- Donc, il est facile d'utiliser Ruby et `MiniTest`... pour faire des tests d'acceptation... comme on le verra dans quelques minutes

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*»

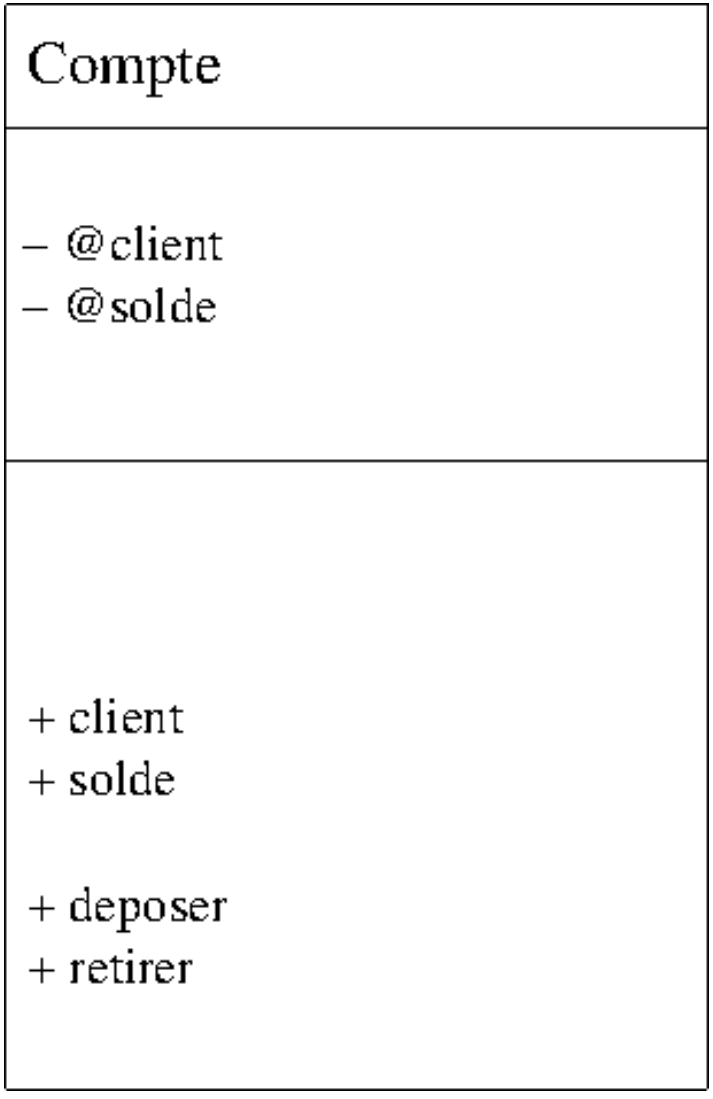
```
describe "#foo"
  it "blah blah blah" do
    ...
    actual.must_equal expected
  end
end
```

Note : `test_helper.rb` (version révisée) utilise une approche hybride !

Exemple : Classe à tester = `Compte` bancaire simple


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

  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`[42]



4.1 Approche avec assertions

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
    solde_avant_depot = @c.solde

    @c.deposer( 100 )
    assert_equal (solde_avant_depot + 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
    solde_avant_depot = @c.solde

    @c.deposer( 100 )
    assert_equal (solde_avant_depot + 100), @c.solde
  end
  ...
```

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


```
  ...
  def test_retirer_deduit_le_montant_desire_lorsque_ne_depasse
    solde_avant_depot = @c.solde

    @c.retirer( 50 )
    assert_equal (solde_avant_depot - 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

4.2 Approche avec *expectations*

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 : 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`)

```
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
```

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_avant_depot = @c.solde

      @c.deposer( 100 )
      @c.solde.must_equal (solde_avant_depot + 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_avant_depot = @c.solde

      @c.deposer( 100 )
      @c.solde.must_equal (solde_avant_depot + 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_avant_depot = @c.solde

      @c.deposer( 100 )
      @c.solde.must_equal (solde_avant_depot + 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_avant_depot = @c.solde

    @c.retirer( 50 )
    @c.solde.must_equal (solde_avant_depot - 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

Dans l'exemple précédent, on remarque que l'ordre d'exécution des tests n'est pas le même que l'ordre des définitions. C'est bien le comportement qu'on retrouve dans plusieurs cadres de tests... dans le but d'assurer l'indépendance des tests.

Dans le cas de `MiniTest`, c'est le rôle du `seed` de définir le germe pour la génération pseudo-aléatoire de l'ordre d'exécution des tests.

Si on veut à tout prix que l'ordre d'exécution respecte l'ordre de définition, alors il faut utiliser la commande suivante au début du programme de test :

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

5. Le principe d’inversion des dépendances et l’injection de dépendances

Propriétés 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
  $\Rightarrow$
  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.

Propriétés 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
  $\Rightarrow$
  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.

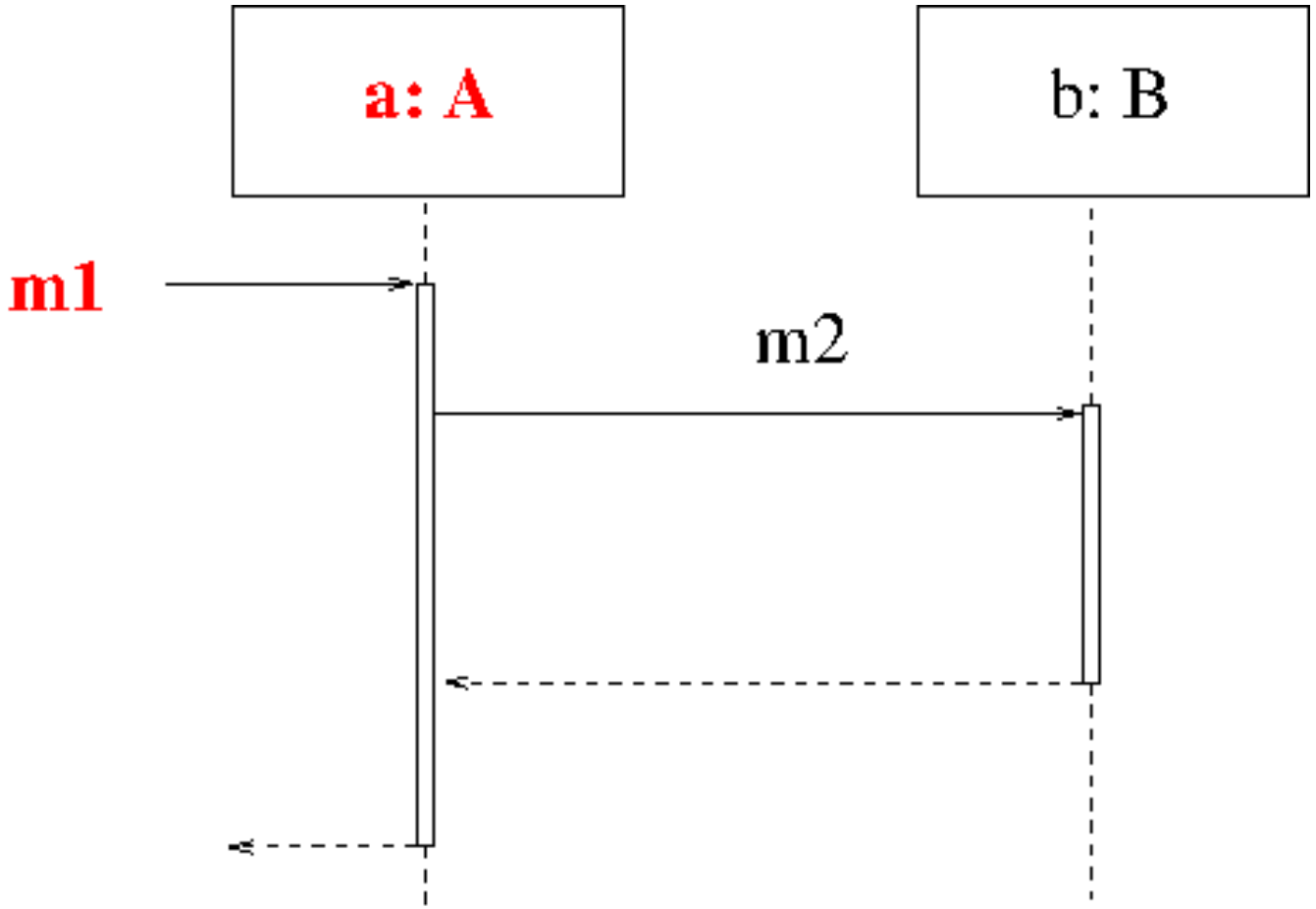
Propriétés des tests unitaires = Indépendants les uns des autres, donc exécutés en «isolation» !

- Les tests unitaires sont supposés être faits «de façon indépendante» pour chaque module, classe, composant
  $\Rightarrow$
  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.

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



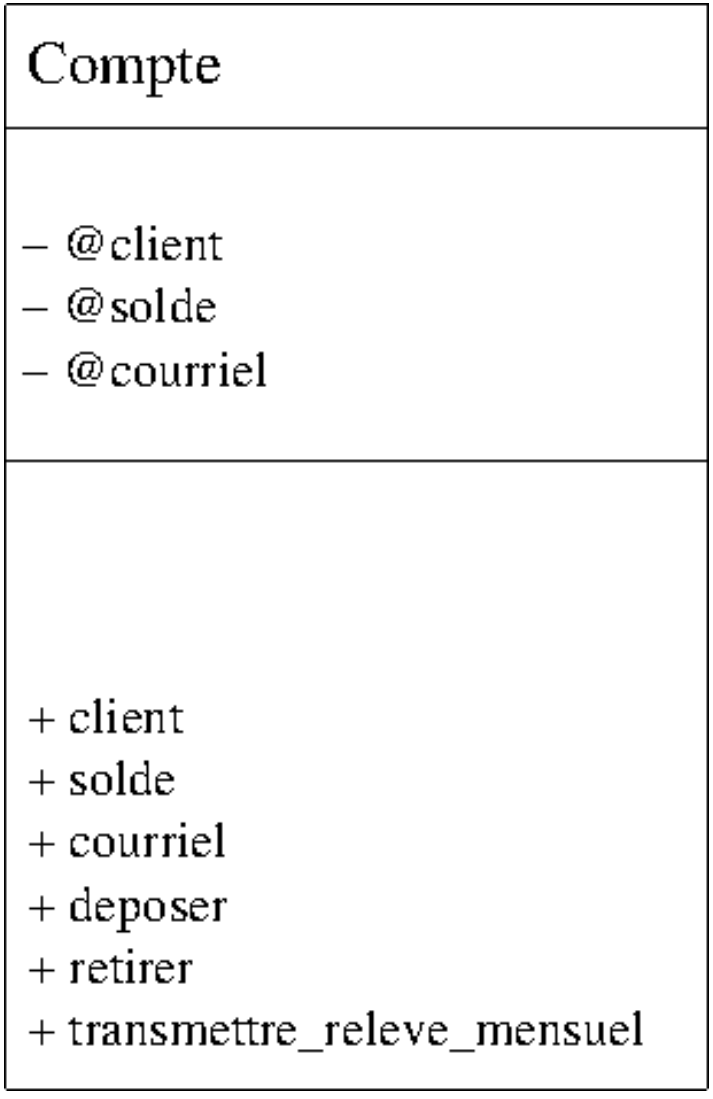
On veut tester la méthode `m1` de la classe `A`...

Exemple : Envoi de relevés mensuels par courriel

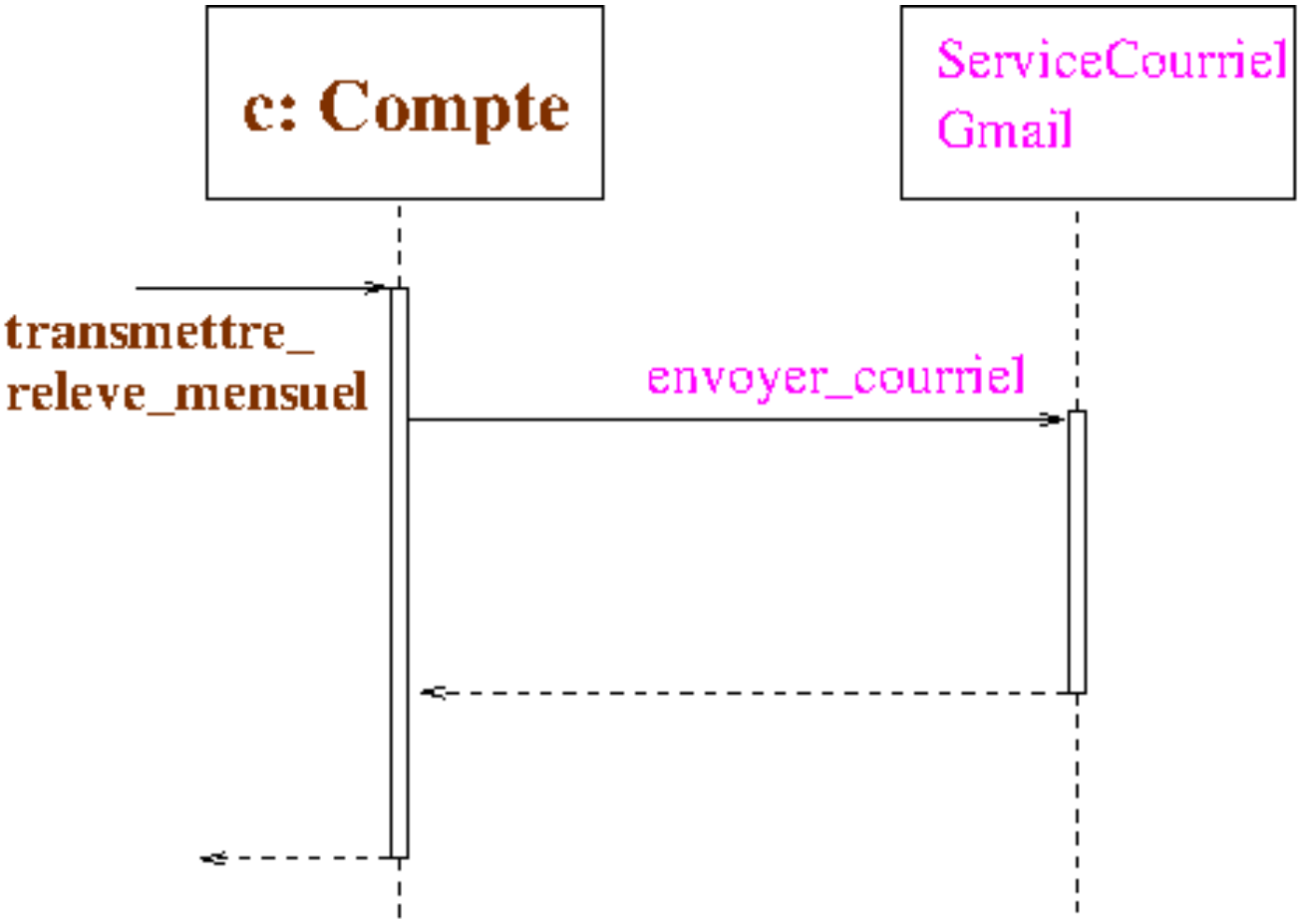
Une nouvelle définition de la classe `Compte`


```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`



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`...

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
```

Le service externe pour l'envoi de courriels


```
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 : 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 ☹

⇒ Test qui 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` !

Injection de dépendances

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»*

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

Registre de services

Injection via un attribut de la classe

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

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
```

Différentes formes d'injection de dépendances

Injection via un attribut de la classe

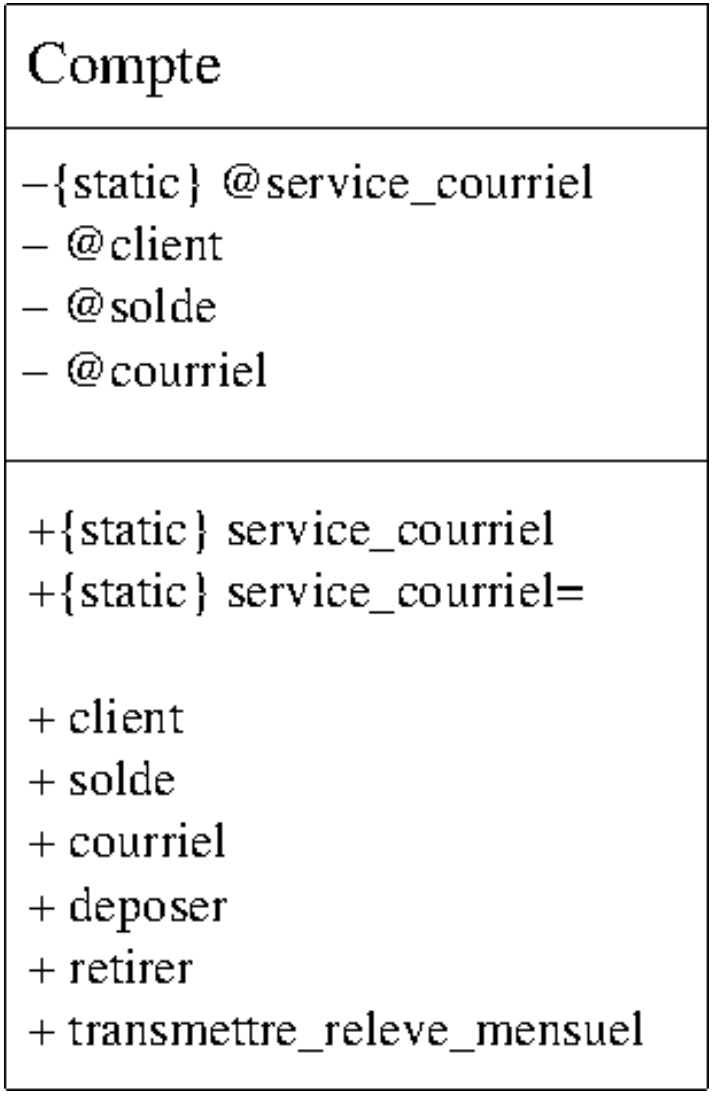
La classe possède une méthode qui permet de spécifier la dépendance, applicable à toutes les instances de la classe.

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

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

Exemple : Envoi de relevés mensuels par courriel (suite)

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



Envoi de courriels avec injection via un attribut de classe

Comment on effectue l'injection

```
# 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 )
...
```

Envoi de courriels avec injection via un attribut de classe

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
end
```

6. Les doublures de tests avec `MiniTest`

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*
- *Dummy objects*
- *Fake objects*
- *Spies*

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

6.1 Les *stubs*

Qu'est-ce qu'une méthode bidon (*stub method*) ?

- *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`

Une classe `Point` simple

La classe possède trois méthodes d'instance — `x`, `y` et `==` — ainsi que 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.
```

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 ) == "?"
```

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
```

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`

Le *stub* pour `envoyer_courriel` va prendre en note les informations transmises si on l'appele (via des variables non locales). On pourra ensuite vérifier que la méthode a bien été appelée.

```
# Variables non-locales au bloc du lambda,
# qui seront modifiees si le lambda est appele.
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 eu lieu.
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. Or, 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

6.2 Les objets *mocks*

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

? ?

Une tentative de test pour `transmettre_releve_mensuel` vue précédemment

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
```

Comment peut-on vérifier — sans aller voir nos courriels — que `transmettre_releve_courriel` a fait son travail ?

Une tentative de test pour `transmettre_releve_mensuel` vue précédemment

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
```

Comment peut-on vérifier — sans aller voir nos courriels — que `transmettre_releve_courriel` a fait son travail ?

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

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`

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*

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

```
mock = MiniTest::Mock.new

# Si un appel :do_something_with(some_obj, true)
# est recu, alors on retourne true.
some_obj = 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 a un appel a foo(_, num1, "abc")
#    -- avec 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 est effectué

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

  # Exercise
  mock.foo  # => 42

  # Verify.
  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… et avec les bons arguments

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( 99, :abc )
end
```

```
# Running:
E
[...]
  1) Error:
Mocks#test_0003_appelle foo avec les mauvais arguments:
MockExpectationError: mocked method :foo called with\
                          unexpected arguments [99, :abc]
    mocks.rb:27:in `block (2 levels) in <main>`

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

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
```

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
```

6.3 Autres formes de doublures de tests

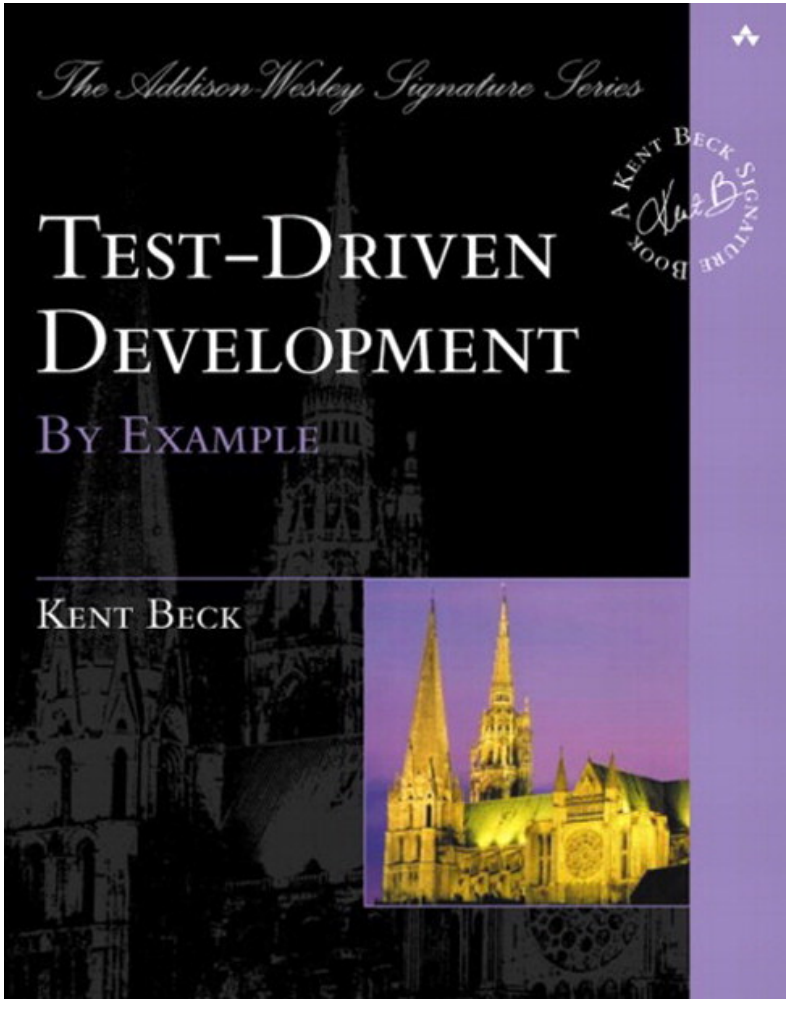
Autres sortes de doublures de tests

- *Dummy objects are passed around but never actually used. Usually they are just used to fill parameter lists.*
- *Fake objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an InMemoryTestDatabase is a good example).*
- *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`

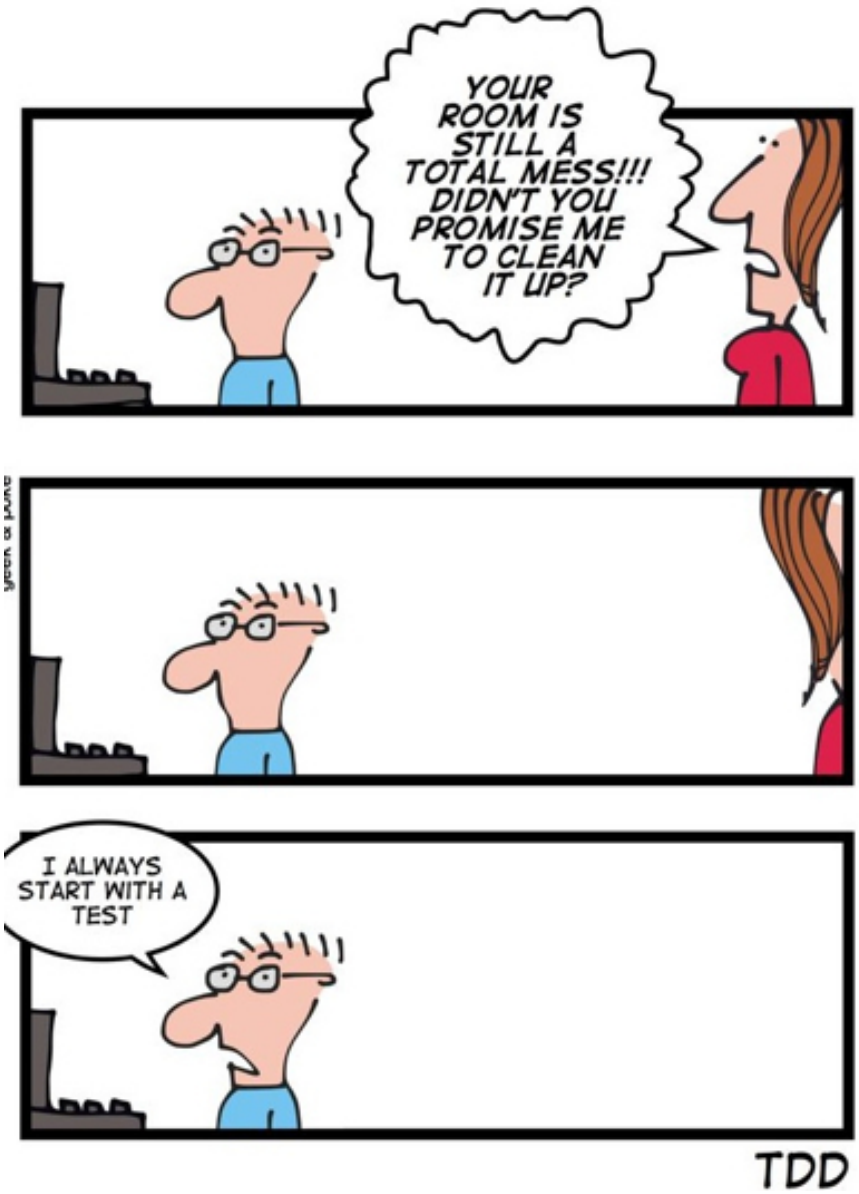
7. Les tests unitaires et l'approche TDD

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 =

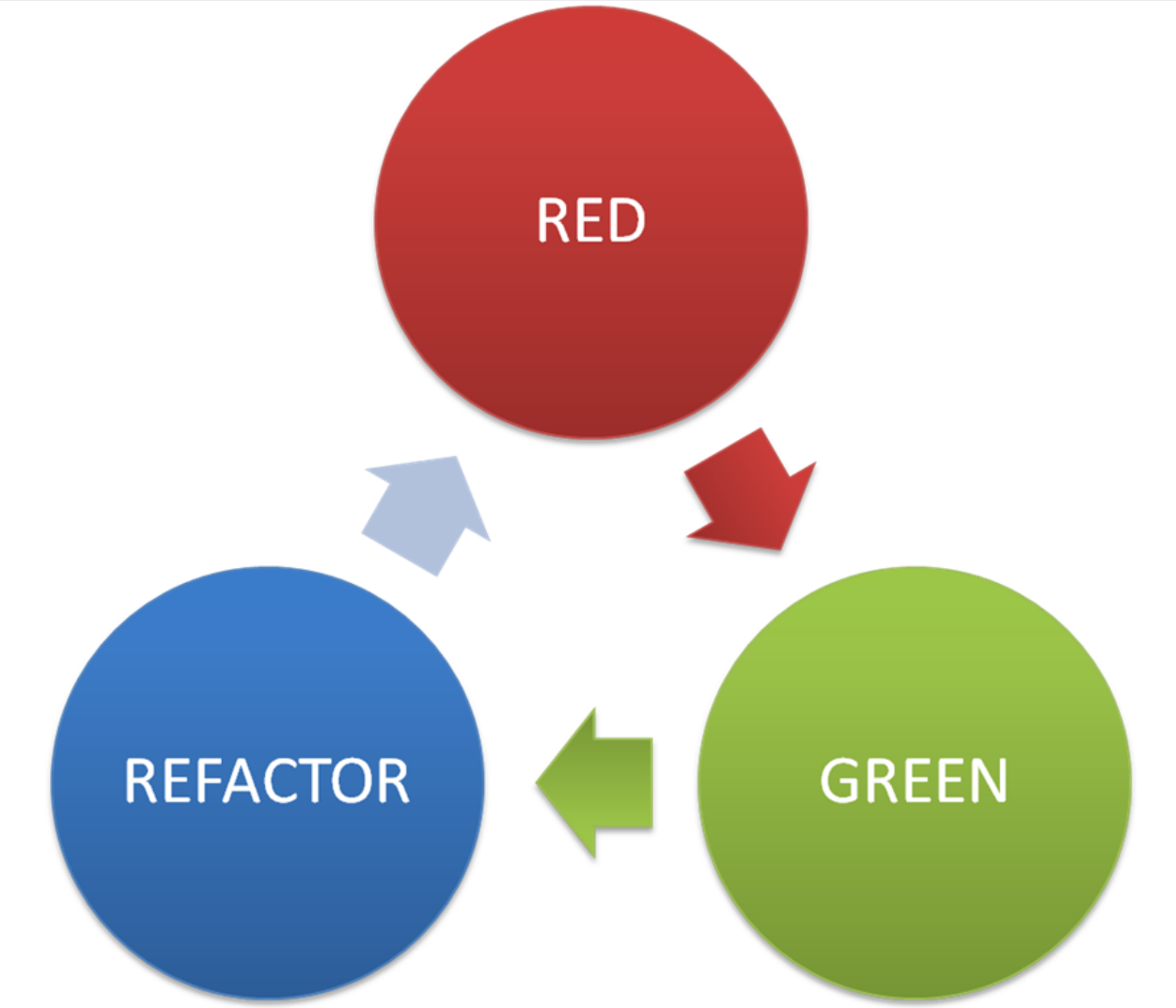
Qu’est-ce que le TDD ?

- TDD = *Test Driven Development*

- Origine = une règle de base de XP =

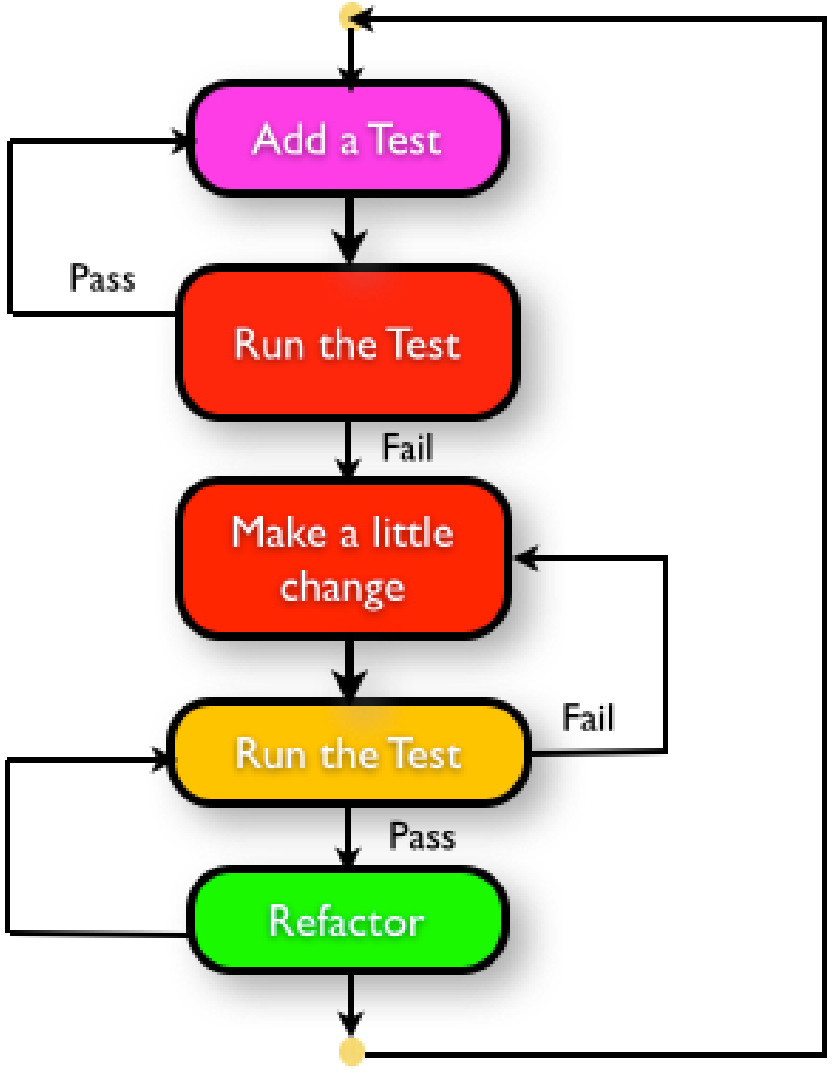
«Code the unit test first !»

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

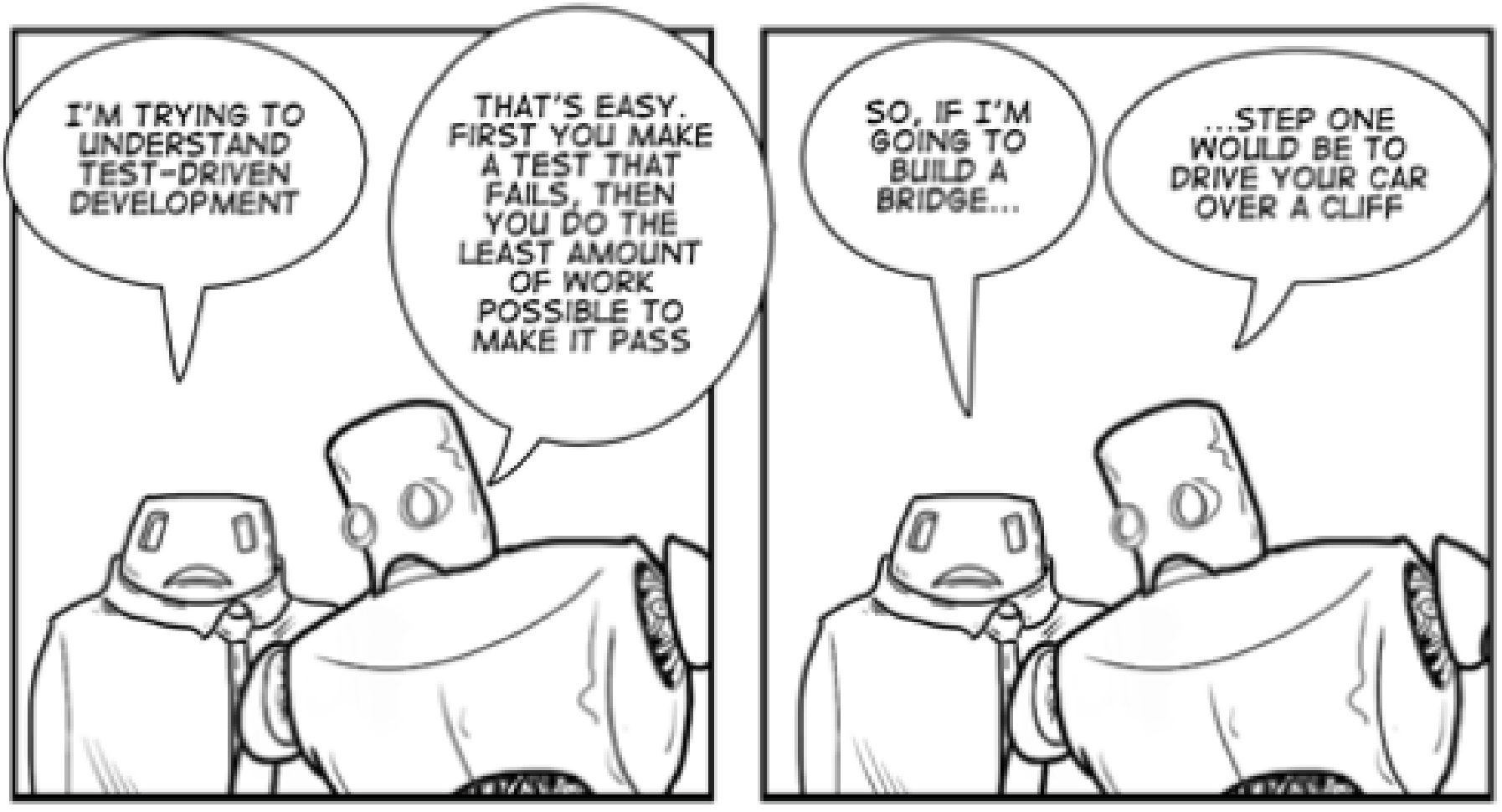
- On est certain d'avoir des tests !

- 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.*

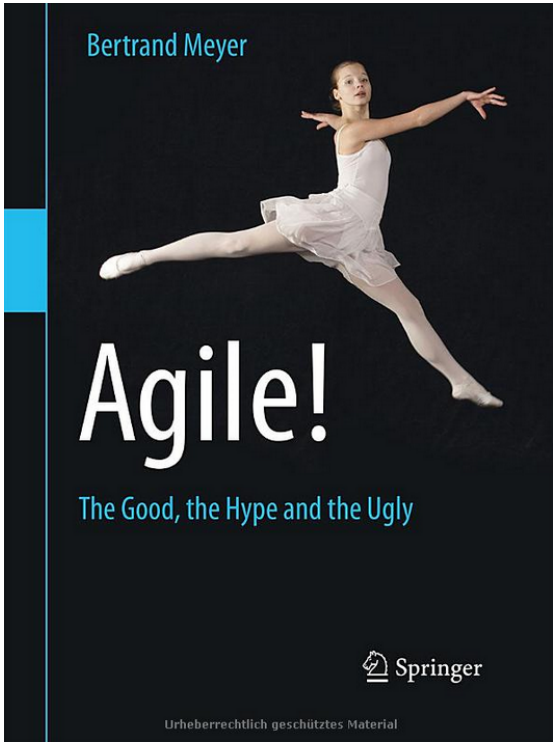
2. *You are not allowed to write any more of a unit test than is sufficient to fail ; and compilation failures are failures.*

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.

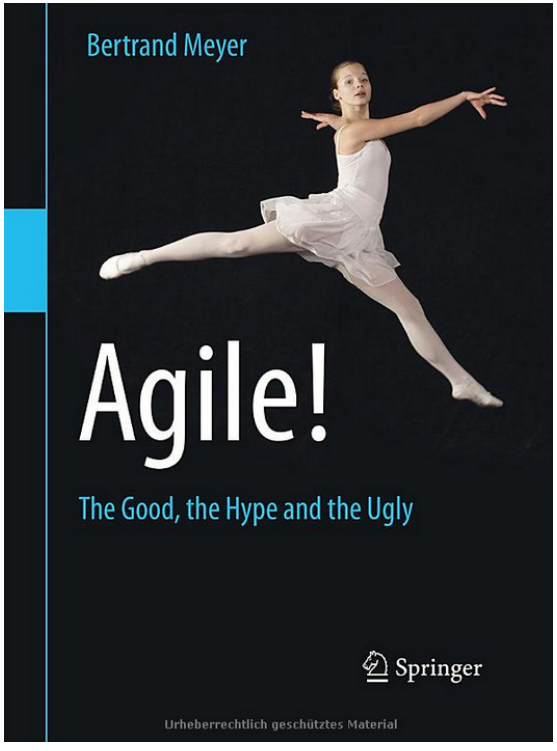
The real insight [is] the idea that any new code must be accompanied by new tests. It is not even critical that the code should come only after the test [. . .] : **what counts is that you never produce one without the other***.»*



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

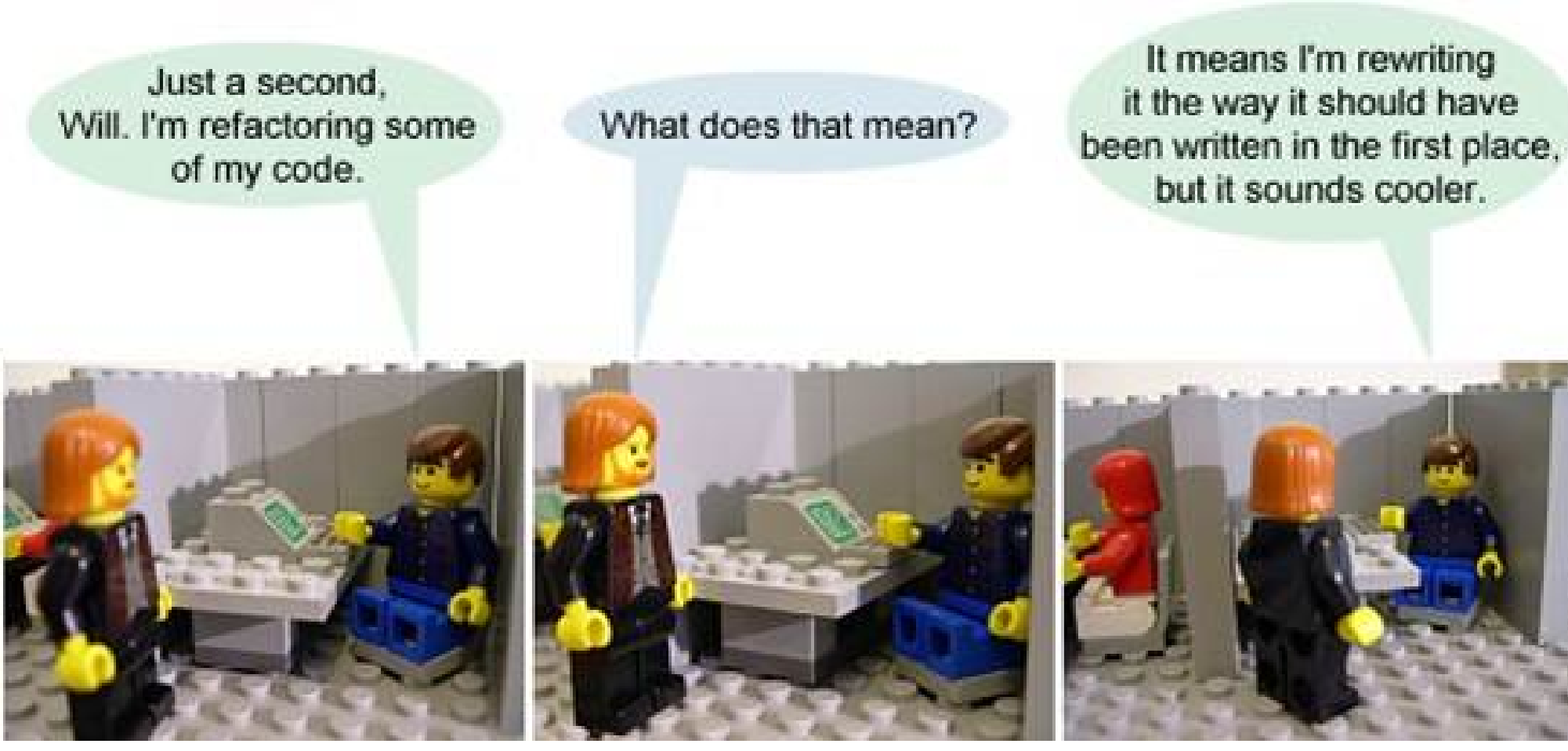
Un avantage d'avoir des tests et du *self-testing code*

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

⇒ 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.

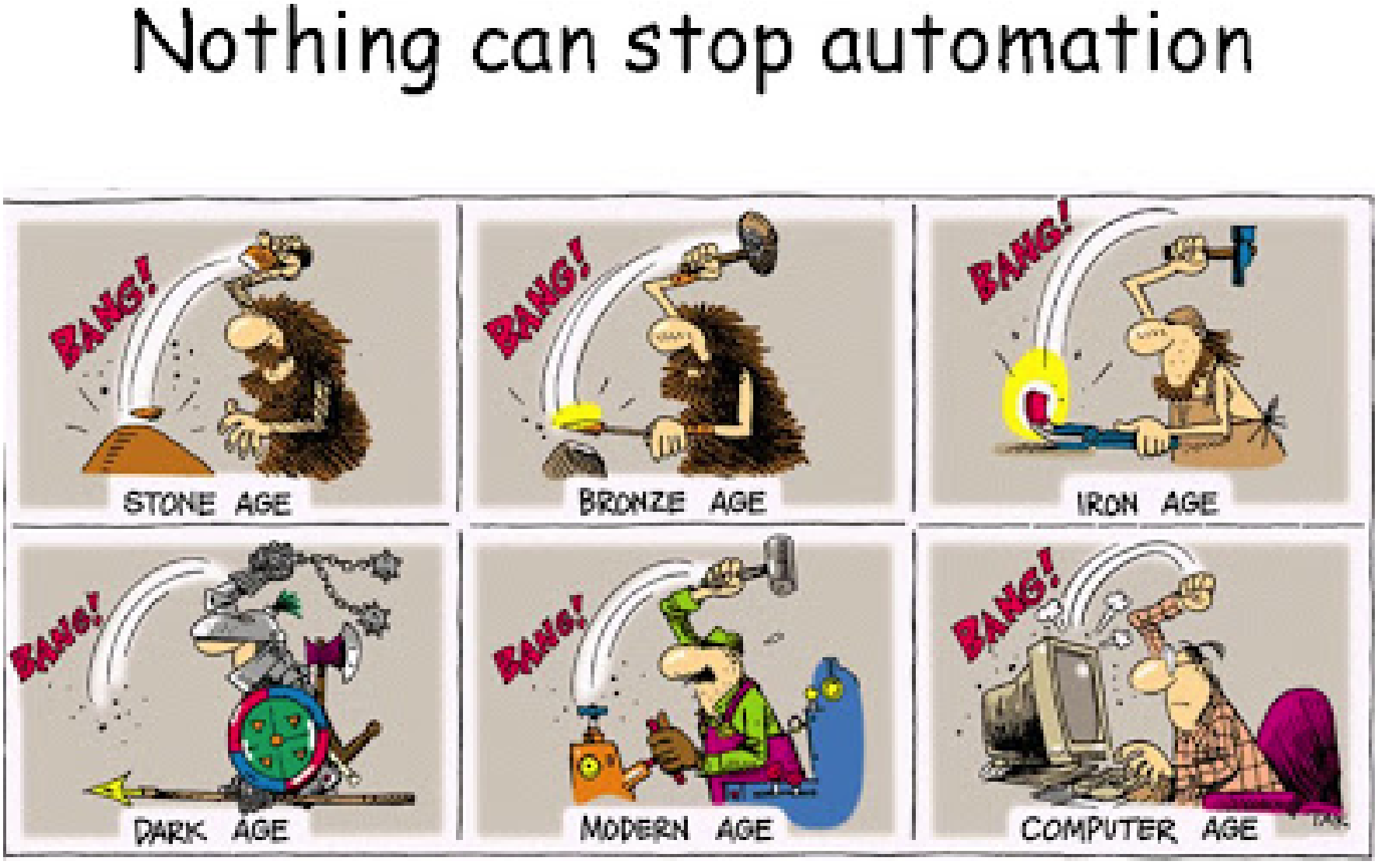
Il est important d'avoir une discipline de travail qui est «professionnelle»

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

[53m15s à 54m42s]
[59m00s à 1h00m40s]

8. Les tests d’acceptation et l’approche BDD

On peut automatiser les tests unitaires, mais aussi les tests d'acceptation !



8.1 Comment peut-on spécifier des tests d’acceptation ?

Récit utilisateur vs. Scénario utilisateur

User story (récit utilisateur)

A user story is a brief statement that identifies the user and her need.

User scenario (scénario utilisateur)

A user scenario expands upon your user stories by including details about how a system might be interpreted, experienced, and used. [. . .] Your scenarios should anticipate the user's goal, specify any assumed knowledge, and speculate on the details of the user's interaction experience.

Source: `https://www.newfangled.com/how-to-tell-the-users-story/`

Caractéristiques des récits utilisateurs

The general guidelines for the user stories themselves is that they must be testable, be small enough to implement in one iteration, and have business value.

Source*: «Engineering Software as a Service», Fox & Patterson*

Gabarit souvent utilisé pour les récits utilisateurs

User stories are short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system.

They typically follow a simple template :

```
As a <type of user>,
I want <some goal>
so that <some reason>.
```

Source: *https://www.mountaingoatsoftware.com/agile/user-stories*

Gabarit souvent utilisé pour les récits utilisateurs

User stories are short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system.

They typically follow a simple template :

```
As a <type of user>,
I want <some goal>
so that <some reason>.
```

Source: *https://www.mountaingoatsoftware.com/agile/user-stories*

Format «Connextra»

As a <role>, I want <goal/desire> so that <benefit>.

Source: http://guide.agilealliance.org/guide/rolefeature.html

Rôle des scénarios utilisateurs = Décrire les conditions d'acceptation pour un récit utilisateur

*This, then, is the role of a Story. It has to be a description of a requirement and its business benefit, and a set of criteria by which we all agree that it is "**done**".*

[The acceptance criteria] are presented as Scenarios.

Source: `http://dannorth.net/whats-in-a-story/`

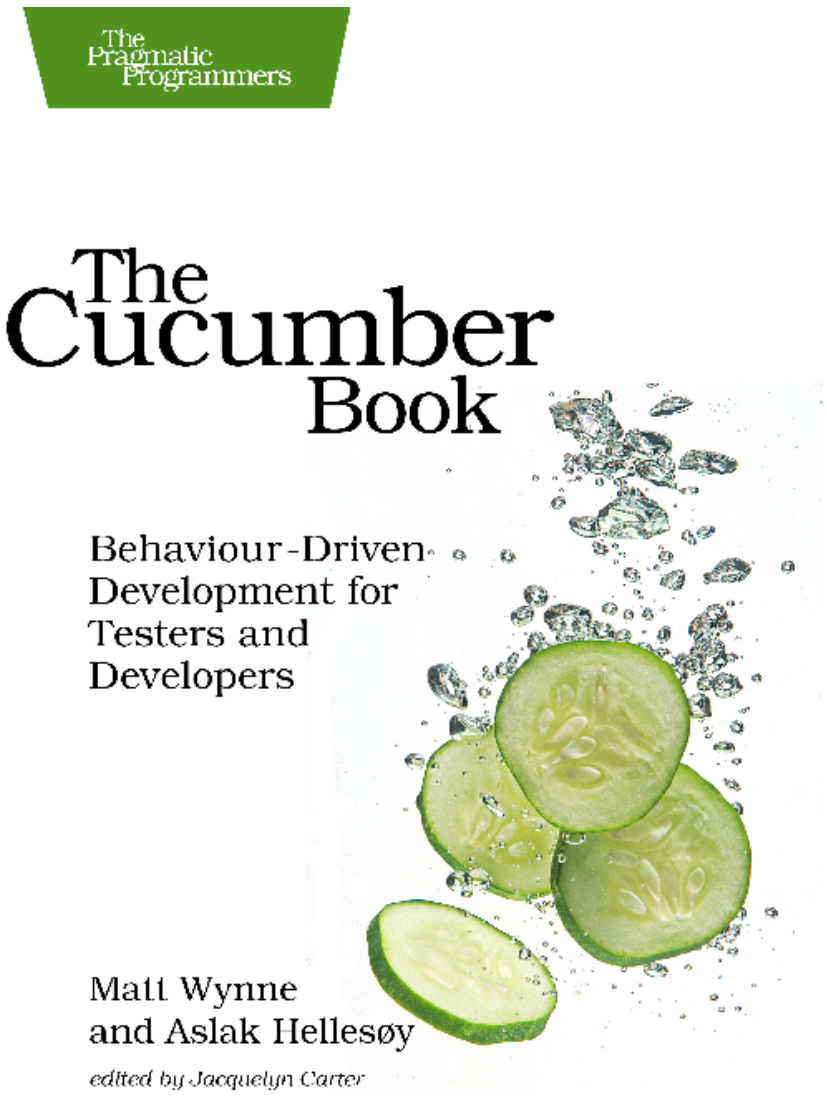
Gabarit souvent utilisé pour les scénarios utilisateurs

```
Scenario: Name of the scenario
 Given some initial context
 When an event occurs
 Then ensure some outcomes
```

Source: Outil `JBehave`, North et Matts, 2004

8.2 Exécution automatique de tests d'acceptation décrits pas des scénarios utilisateurs : L'outil `cucumber`

Cucumber (Hellesoy, 2008)

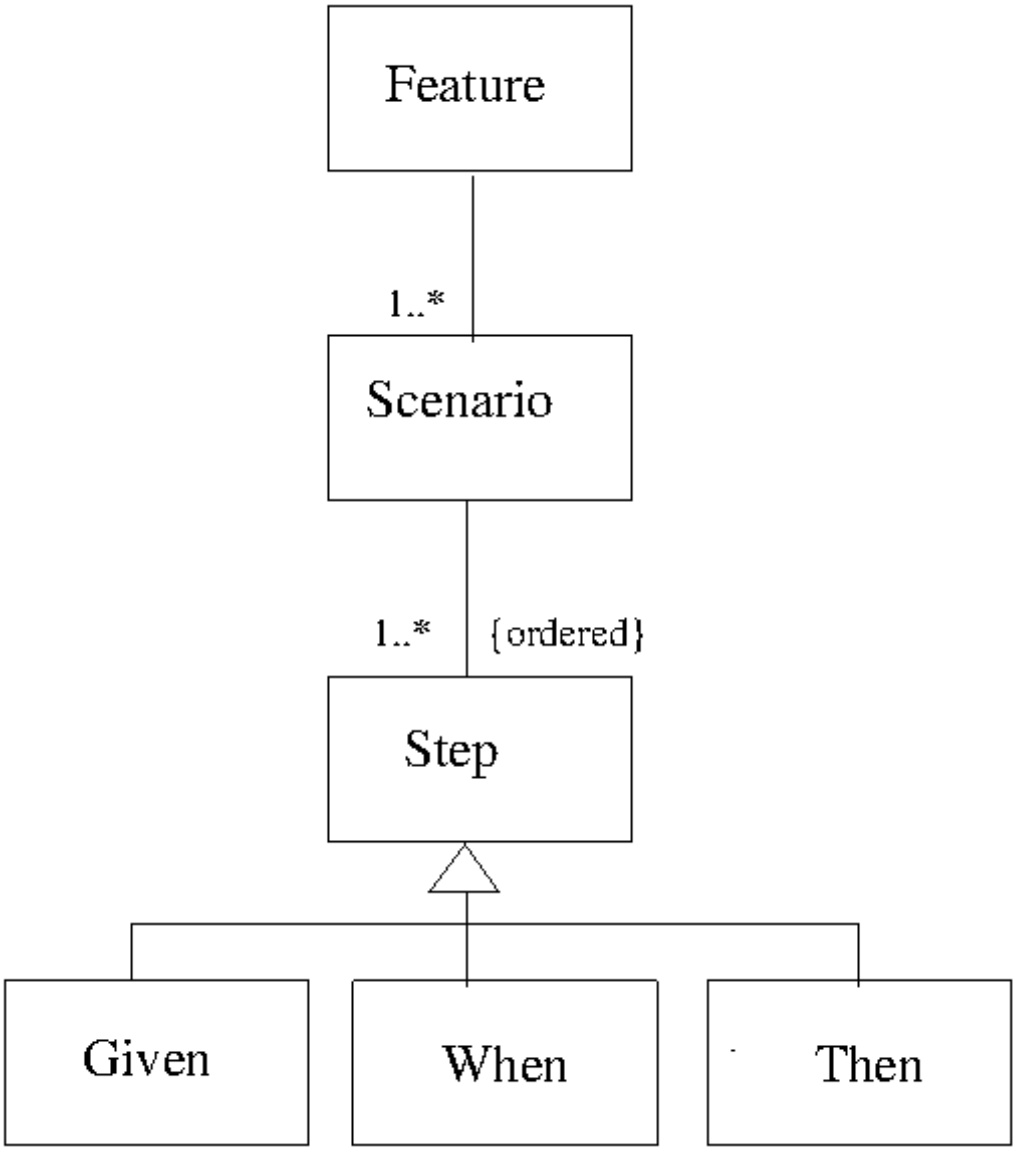


Pour la spécification et l'exécution des tests d'acceptation !

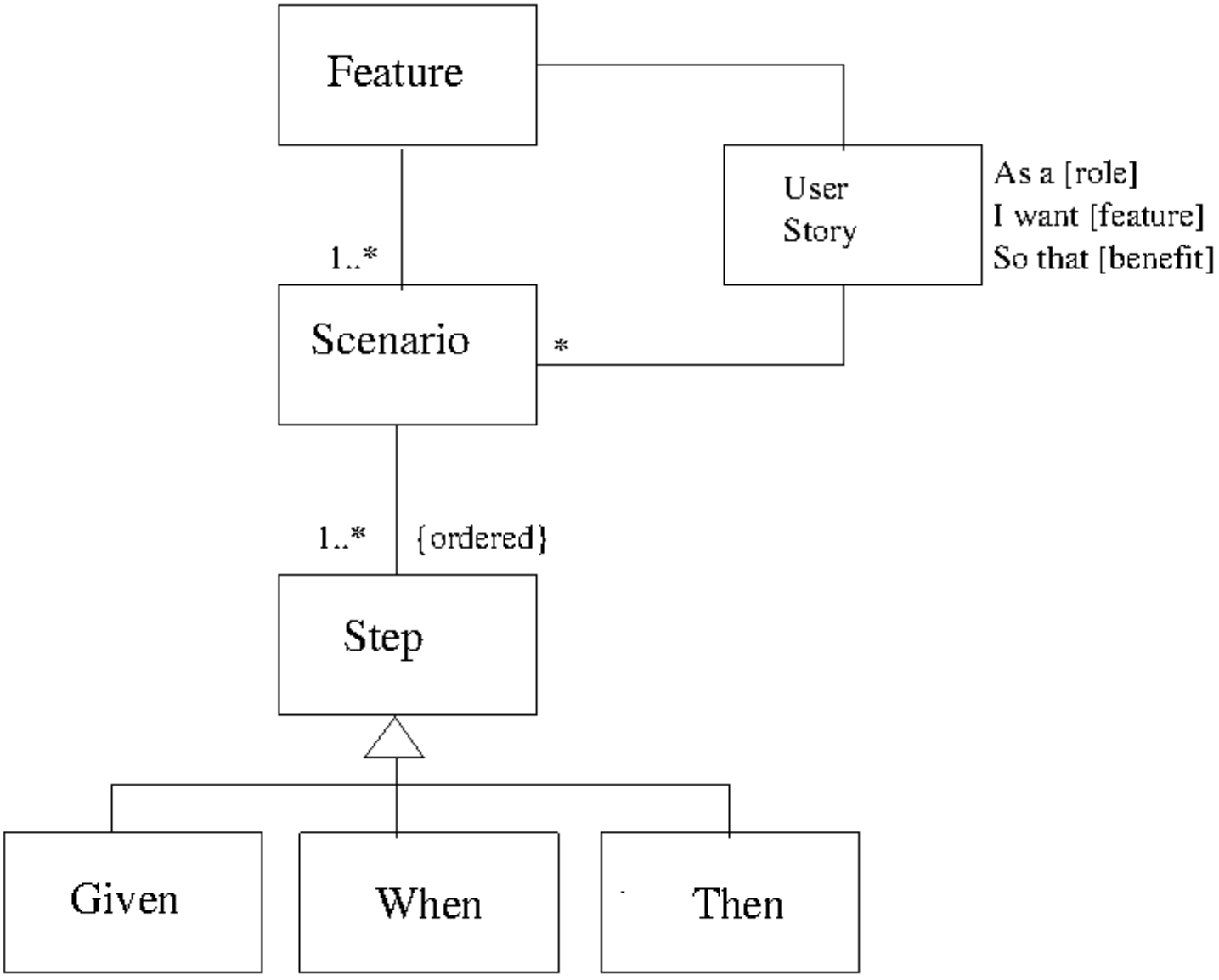
Lorsqu'on utilise `cucumber`, les récits et scénarios utilisateurs sont organisés par *feature*


```
Feature
    User Story

    Scenario 1
        .
        .
        .
    Scenario n
```

Lorsqu'on utilise `cucumber`, les récits et scénarios utilisateurs sont organisés par *feature*



Lorsqu'on utilise `cucumber`, les récits et scénarios utilisateurs sont organisés par *feature*



Les trois sortes de `step`s qui composent un scénario Cucumber

Given

Identifie l'état courant/initial, dans lequel le scénario va s'appliquer.

When

Identifie l'action ou l'événement qui déclenche le scénario.

Then

Identifie les conséquences de l'action.

Les trois sortes de `steps` qui composent un scénario Cucumber

The purpose of givens is to put the system in a known state before the user (or external system) starts interacting with the system (in the When steps).

The purpose of When steps is to describe the key action the user performs [. . .].

The purpose of Then steps is to observe outcomes. The observations should be related to the business value/benefit in your feature description. The observations should also be on some kind of output — that is something that comes out of the system (report, user interface, message)[. . .].

Source: `https://github.com/cucumber/cucumber/wiki/Given-When-Then`

Les trois sortes de `steps` qui composent un scénario Cucumber

Given-When-Then is a style of representing tests — or as its advocates would say — specifying a system's behavior using SpecificationByExample.

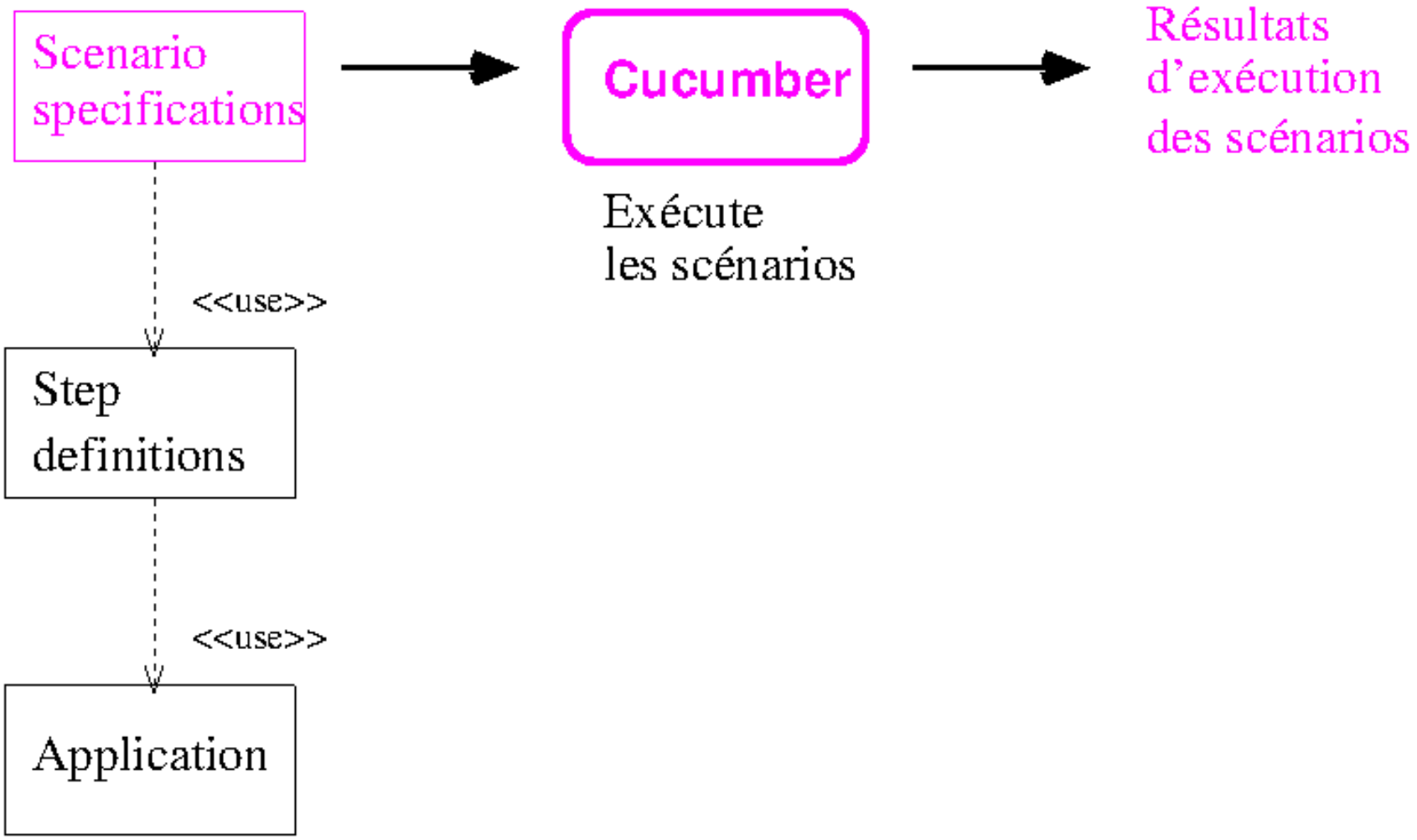
The given part describes the state of the world before you begin the behavior you're specifying in this scenario. You can think of it as the pre-conditions to the test.

The when section is that behavior that you're specifying.

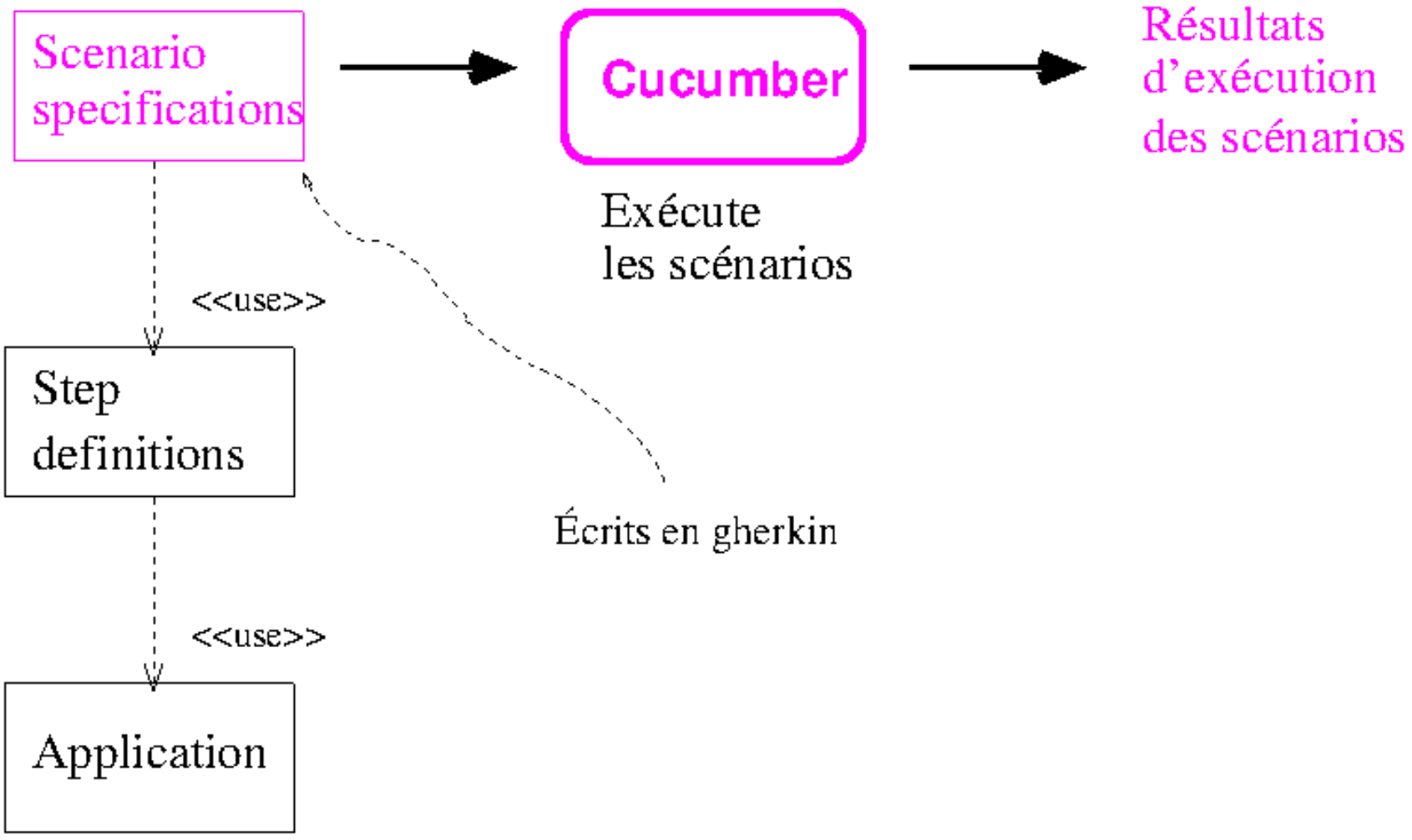
Finally the then section describes the changes you expect due to the specified behavior.

Source: `http://martinfowler.com/bliki/GivenWhenThen.html`

Cucumber : Principe général de fonctionnement



Cucumber : Principe général de fonctionnement



Exemple avec `cucumber` et `aruba`

Tests d'acceptation pour `wc` = fichier `features/wc.feature`


```
$ cat features/wc.feature
Feature: Compter le nombre de mots avec wc

  En tant que programmeur
  Je veux pouvoir utiliser wc
  Afin de compter le nombre de lignes, de mots
       ou de caracteres

    ...
```

Note : `aruba` est un DSL (*Domain Specific Language*) pour spécifier des tests d'acceptation pour une application exécutée en mode «ligne de commandes».

Exemple avec `cucumber` et `aruba`

Tests d'acceptation pour `wc` = fichier `features/wc.feature` (suite)


```
Scenario: Fichiers avec plusieurs lignes et mots
 Given a file "test1.txt" with:
 """
 abc def
 ghi j
 k

 """

 When I run `/usr/bin/wc -lw test1.txt`

 Then the output should match /^\s+3\s+5\s+test1.txt$/
 And the exit status should be 0

 ...
```

Exemple avec `cucumber` et `aruba`

Tests d'acceptation pour `wc` = fichier `features/wc.feature` (suite)


```
Scenario: Fichier vide
   Given an empty file "test1.txt"

   When I run `/usr/bin/wc -lw test1.txt`

   Then the output should match /^\s*0\s+0\s*/
   And the exit status should be 0
```

Exemple avec `cucumber` et `aruba`

Tests d'acceptation pour `wc` = fichier `features/wc.feature` (suite)

```
Scenario: Fichier inexistant
  Given a file "foo.txt" does not exist

  When I run `/usr/bin/wc -lw foo.txt`

  Then the output should match /Aucun.*fichier/
  And the exit status should not be 0
```

Exemple avec `cucumber` et `aruba`

Exécution des tests d'acceptation où les trois scénarios s'exécutent avec succès — format «`progress`»

```
$ cucumber --format=progress --color
............

3 scenarios (3 passed)
12 steps (12 passed)
0m0.344s
```

Exemple avec `cucumber` et `aruba`

Exécution des tests d'acceptation où les trois scénarios s'exécutent avec succès — format «`pretty`»

```
(master) ExempleCucumber@linux $ cucumber --color
Feature: Compter le nombre de mots avec wc

  En tant que programmeur
  Je veux pouvoir utiliser wc
  Afin de compter le nombre de lignes, de mots et/ou de caracteres

  Scenario: Fichiers avec plusieurs lignes et mots        # features/wc.feature:7
    Given a file "test1.txt" with:                        # aruba-0.8.1/lib/aruba/cucumber.rb:42
      """
      abc def
      ghi j
      k

      """
    When I run `/usr/bin/wc -wl test1.txt`                # aruba-0.8.1/lib/aruba/cucumber.rb:127
    Then the output should match /^\s+3\s+5\s+test1.txt$/ # aruba-0.8.1/lib/aruba/cucumber.rb:214
    And the exit status should be 0                       # aruba-0.8.1/lib/aruba/cucumber.rb:235

  Scenario: Fichier vide                    # features/wc.feature:22
    Given an empty file "test1.txt"         # aruba-0.8.1/lib/aruba/cucumber.rb:55
    When I run `/usr/bin/wc -wl test1.txt`  # aruba-0.8.1/lib/aruba/cucumber.rb:127
    Then the output should match /0\s+0/    # aruba-0.8.1/lib/aruba/cucumber.rb:214
    And the exit status should be 0         # aruba-0.8.1/lib/aruba/cucumber.rb:235

  Scenario: Fichier inexistant                      # features/wc.feature:30
    Given a file "foo.txt" does not exist           # aruba-0.8.1/lib/aruba/cucumber.rb:92
    When I run `/usr/bin/wc -wl foo.txt`            # aruba-0.8.1/lib/aruba/cucumber.rb:127
    Then the output should match /Aucun.*fichier/   # aruba-0.8.1/lib/aruba/cucumber.rb:214
    And the exit status should not be 0             # aruba-0.8.1/lib/aruba/cucumber.rb:239

3 scenarios (3 passed)
```

Exemple avec `cucumber` et `aruba`

Exécution des tests d'acceptation où un des scénarios échoue — format «`progress`»

```
$ cucumber --format=progress --color
......F-....

(::) failed steps (::)

expected "0 0 test1.txt\n" to match /0\s+1/m
Diff:
@@ -1,2 +1,2 @@
-/0\s+1/m
+0 0 test1.txt
(RSpec::Expectations::ExpectationNotMetError)
features/wc.feature:27:in `Then the output should match /0\s+1

Failing Scenarios:
cucumber features/wc.feature:22 # Scenario: Fichier vide

3 scenarios (1 failed, 2 passed)
12 steps (1 failed, 1 skipped, 10 passed)
0m0.365s
```

Le DSL de `cucumber`, `gherkin`, supporte de nombreux langages, dont le français


```
Scénario: J'ai assez d'argent dans mon compte
  Soit mon compte a un solde de 200 dollars

  Quand je retire 200 dollars

  Alors je recois 200 dollars
  Et le solde de mon compte est de 0 dollars


Scénario: Je n'ai pas assez d'argent dans mon compte
  Soit mon compte a un solde de 200 dollars

  Quand je retire 500 dollars

  Alors je recois un message d'erreur
  Et le solde de mon compte est de 200 dollars
```

Le DSL de `cucumber`, `gherkin`, permet de définir des familles de cas de tests


```
Scenario Outline: J'ai assez d'argent dans mon compte
  Given mon compte a un solde de <solde_initial> dollars

  When je retire <montant> dollars

  Then je recois <montant> dollars
  And le solde de mon compte est de <solde_final> dollars

  Scenarios:
    | solde_initial | montant | solde_final |
    | 200           | 50      | 150         |
    | 200           | 200     | 0           |
```

```
<<~/SeminaireTBDDD/Compte@MacBook>> $ cucumber
Feature: Retrait d'un montant d'un compte
  En tant que responsable d'un compte
  Je veux pouvoir retirer un montant de mon compte
  Afin d'avoir de l'argent comptant sous la main

  Scenario Outline: J'ai assez d'argent dans mon compte     # features/retirer.feature:6
    Given mon compte a un solde de <solde_initial> dollars  # features/compte_steps.rb:29
    When je retire <montant> dollars                        # features/compte_steps.rb:19
    Then je recois <montant> dollars                        # features/compte_steps.rb:42
    And le solde de mon compte est de <solde_final> dollars # features/compte_steps.rb:38

    Scenarios:
      | solde_initial | montant | solde_final |
      | 200           | 50      | 150         |
      | 200           | 200     | 0           |

  Scenario: Je n'ai pas assez d'argent dans mon compte # features/retirer.feature:20
    Given mon compte a un solde de 200 dollars         # features/compte_steps.rb:29
    When je retire 500 dollars                         # features/compte_steps.rb:19
    Then je recois un message d'erreur                 # features/compte_steps.rb:46
    And le solde de mon compte est de 200 dollars      # features/compte_steps.rb:38

3 scenarios (3 passed)
12 steps (12 passed)
0m0.018s
<<~/SeminaireTBDDD/Compte@MacBook>> $
```

Le DSL de `cucumber`, `gherkin`, permet de définir des valeurs qui sont des suites de lignes

Appelé des `docstring` (comme en Python)

```
Scenario: J'effectue plusieurs retraits
  Given mon compte a un solde de 100.00$

  When je retire les montants suivants:
    """
    20.20$
    10.05$
    30.40$
    """

  Then le solde de mon compte est de 39.35$
```

Le DSL de `cucumber`, `gherkin`, permet de définir des valeurs qui sont des suites de lignes

L'objet reçu est une chaine, avec un saut de ligne qui sépare chaque ligne

```
When(/^je retire les montants suivants:$/) do |string|
  string.split("\n").map { |c| c.chop.to_f }.each do |montant|
    @c.retirer montant
  end
end
```

Donc : il n'y a pas de saut de ligne après le dernier item !

Le DSL de `cucumber`, `gherkin`, permet de définir des tables de données


```
Scenario: J'effectue plusieurs operations
  Given mon compte a un solde de 100.00$

  When j'effectue les operations suivantes:
    | depot    |  50.00$ |
    | retrait  |  80.00$ |
    | retrait  |  20.00$ |
    | retrait  |  10.00$ |
    | depot    | 200.00$ |

  Then le solde de mon compte est de 240.00$
```

Le DSL de `cucumber`, `gherkin`, permet de définir des tables de données

L'objet reçu est un `DataTable`

```
When(/^j'effectue les operations suivantes:$/) do |table|
  # table is a Cucumber::MultilineArgument::DataTable
  table.raw.each do |op, montant|
    montant = montant.chop.to_f
    if op == "depot"
      @c.deposer montant
    elsif op == "retrait"
      @c.retirer montant
    else
      fail "*** Operation inconnue: #{operation}"
    end
  end
end
```

Pour plus de détails sur les `DataTable` :

`http://www.rubydoc.info/gems/cucumber/Cucumber/MultilineArgument/DataTable`

Généralement, les scénarios sont écrits de façon indépendante des détails d'IPM


```
Scénario: J'emprunte plusieurs livres
  Soit la BD existe et est vide

  Quand "nom1" ["@"] emprunte "titre1" ["auteurs1"]
  Et "nom2" ["@"] emprunte "titre2" ["auteurs2"]
  Et "nom3" ["@"] emprunte "titre3" ["auteurs3"]

  Alors il y a 3 emprunts
  Et l'emprunteur de "titre1" est "nom1"
  Et l'emprunteur de "titre2" est "nom2"
  Et l'emprunteur de "titre3" est "nom3"
```

Notamment : Ce même scénario pourrait être utilisé tant pour un script `biblio.sh` (script `bash` en ligne de commande), que pour une mises en œuvre Ruby elle aussi, en ligne de commandes, ou même une application Web !

Références

K. Beck.

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

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.

J. Rasmusson.

The Agile Samurai—How Agile Masters Deliver Great Software.
The Pragmatic Bookshelf, 2010.

V. Subramaniam and A. Hunt.

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

M. Wynne and A. Hellesoy.

The Cucumber Book : Behaviour-Driven Development for Testers and Developers.
The Pragmatic Bookshelf, 2012.

A. Un aperçu des tests d’acceptation du devoir 1

Les tests fournis avec le devoir 1 sont des tests d'acceptation

Chaque test vérifie le bon fonctionnement d'une commande, mais en exécutant le script `gv.sh` dans son ensemble — de bout en bout :

1. Définit le dépôt de données — i.e., le fichier pour la liste de vins utilisée par le test
2. Appelle le script `gv.sh` avec divers arguments et options
3. Vérifie que…
   a. la sortie émise sur `stdout` est celle attendue
   b. la sortie émise sur `stderr` est celle attendue
   c. le code de statut retourné est celui attendu
   d. le dépôt de données, le cas échéant, a été modifié correctement

Ces tests sont mis en œuvre à l'aide de méthodes Ruby et avec `MiniUnit`, un cadre de tests unitaires

Méthodes définies dans `Tests/test_helper.rb`

```
avec_fichier( nom_fichier, lignes = [], conserver = nil )
run_gv( *cmds )
genere_erreur( erreur )
execute_sans_sortie_ou_erreur
genere_sortie( attendu, strict = nil  )
genere_sortie_et_erreur( attendu, erreur )
```

Tests MiniUnit de style RSpec (avec *expectations*)

```
describe "blah blah"
  it_ "blah blah blah" do
    ...
    actual.must_equal expected
  end
end
```

Ces tests sont mis en œuvre à l'aide de méthodes Ruby et avec `MiniUnit`, un cadre de tests unitaires

Méthodes définies dans `Tests/test_helper.rb`

```
avec_fichier( nom_fichier, lignes = [], conserver = nil )
run_gv( *cmds )
genere_erreur( erreur )
execute_sans_sortie_ou_erreur
genere_sortie( attendu, strict = nil  )
genere_sortie_et_erreur( attendu, erreur )
```

Tests MiniUnit de style RSpec (avec *assertions*)

```
describe "blah blah"
  it_ "blah blah blah" do
    ...
    assert_equal expected, actual
  end
end
```

Un cas de test = un appel à la méthode `it_`

Exemples tirés du fichier `Tests/lister_test.rb`

```
it_ "liste un fichier vide" do
  ...
end

it_ "liste, par defaut, tous [...] forme longue" do
  ...
end

it_ "genere erreur [...] inexistant", :intermediaire do
  ...
end
...
```

- Argument #1 (`String`) = Nom du test (`it`/`it_`)
- Argument #2 (`Symbol`, défaut = `:base`) = Niveau du test (`it_`)
  - `:base`
  - `:intermediaire`
  - `:avance`

Dépôt utilisé dans les exemples qui suivent


```
$ cat Tests/4vins.txt
1:10/06/18:rouge:Chianti Classico:2015:Volpaia:26.65:4:Fonce,
2:10/06/18:rouge:Chianti Classico:2014:Volpaia:26.65::
4:03/07/18:blanc:Alsace:2016:Pfaff:16.50::
5:03/07/18:rose:Cotes de Provence:2017:Roseline:18.50:3:Frais,
```

Exemple : Un cas de test qui réussit pour `lister`

(doit s'exécuter sans erreur)

Cas de test du fichier `Tests/lister_spec.rb` utilisant le fichier `Tests/4vins.txt`


```
it_ "liste tous les vins [...] --court", :intermediaire do
  lignes = IO.readlines("Tests/4vins.txt")

  attendu = [
             '1 [26.65$]: Chianti Classico 2015, Volpaia',
             '2 [26.65$]: Chianti Classico 2014, Volpaia',
             '4 [16.50$]: Alsace 2016, Pfaff',
             '5 [18.50$]: Cotes de Provence 2017, Roseline',
            ]

  avec_fichier $DEPOT_DEFAUT, lignes do
    genere_sortie attendu do
      run_gv( 'lister --court' )
    end
  end
end
```

```
$DEPOT_DEFAUT = '.vins.txt' # Variable globale.
```

Méthode `avec_fichier`

Fichier `Tests/test_helper.rb`


```
lignes = IO.readlines("Tests/4vins.txt")
...

avec_fichier $DEPOT_DEFAUT, lignes do
   .
   .
   .
end
```

1. Crée un fichier `.vins.txt` avec les `lignes` indiquées
2. Exécute le code à l'intérieur du `do ... end` (bloc)
3. Supprime le fichier `.vins.txt` (à la sortie du bloc)

Méthode `run_gv`

Fichier `Tests/test_helper.rb`


```
run_gv( 'lister --court' )
```

1. Lance l'exécution du script `gv.sh` avec la commande `lister` et son option `--court`
2. Capture les sorties produites sur `std{out,err}` ainsi que le code de statut
3. Transforme les sorties de `std{out,err}` en tableaux de lignes — sans le caractère «`\n`»
4. Retourne un résultat (triplet) composé des deux tableaux de lignes de résultats et du code de statut

Méthode `run_gv` : Mise en œuvre

Fichier `Tests/test_helper.rb`

```ruby
def run_gv( *cmds )     # Nombre variable d'arguments
  cmd_line = cmds
               .map{ |cmd| "./gv.sh #{cmd}" }
               .join( ' | ' )

  stdout = stderr = waiting_thread = nil
  Open3.popen3( "#{cmd_line}" ) do |i, o, e, w|
    i.close
    stdout = o.readlines.map!(&:chomp!)
    stderr = e.readlines.map!(&:chomp!)
    waiting_thread = w
  end

  [stdout, stderr, waiting_thread.value.exitstatus]
end
```

Note : `chomp!` supprime les «`\n`» des lignes.

Méthode `genere_sortie`

Fichier `Tests/test_helper.rb`


```
genere_sortie attendu do
  .
  .
  .
end
```

1. Exécute le code dans le bloc, qui retourne un triplet :
   i. Tableau des lignes émises sur `stdout` (sans «`\n`»)
   ii. Tableau des lignes émises sur `stderr` (sans «`\n`»)
   iii. Code de statut
2. Vérifie que le 1^{er} résultat (lignes de `stdout`) est celui indiqué par `attendu`
3. Vérifie que le 2^{e} résultat (lignes de `stderr`) est vide
4. Vérifie que le code de statut est 0

Défaut = la vérification du contenu de la sortie se fait de façon indulgente

Indulgent = ignore la casse et les espaces

```
it_ "liste [...] avec l'option --court", :intermediaire do
  lignes = IO.readlines("Tests/4vins.txt")

  attendu =
    [
    '  1   [  26.65$]:  chianti classico 2015,  Volpaia',
    '2 [26.65$]: chianti classico 2014,     Volpaia',
    '  4 [16.50$]:    alsace 2016,   pfaff  ',
    '5     [18.50$]: Cotes de   Provence 2017,  Roseline',
    ]

  avec_fichier $DEPOT_DEFAUT, lignes do
    genere_sortie attendu do
      run_gv( 'lister --court' )
    end
  end
end
```

Mais dans certains cas, la vérification du contenu se fait de façon stricte

Strict = tient compte de la casse et des espaces

```
it_ "inclut les items de la chaine", :intermediaire do
  avec_fichier $DEPOT_DEFAUT, lignes do
    genere_sortie ['Note pour 1 => 4',
                   'Note pour 5 => 3'], :strict do
      run_gv( 'selectionner --bus',
              '- lister --format="Note pour I => n"' )
    end
  end
end
```

Méthode `genere_sortie` : Mise en œuvre

Fichier `Tests/test_helper.rb`

```
def genere_sortie( attendu, strict = nil  )
  out, err, statut = yield

  if strict
    assert_equal attendu, out,
      "*** Assertion echouee: La sortie emise sur stdout [...]
  else
    obtenu = out.map { |l| l.gsub(/\s+/, "").downcase }
    attendu = attendu.map { |l| l.gsub(/\s+/, "").downcase }
    assert_equal attendu, obtenu,
      "*** Assertion echouee: La sortie emise sur stdout [...]
  end
  assert_empty err,
    "*** Assertion echouee: stderr devrait etre vide [...] ***
  assert_equal 0, statut,
    "*** Assertion echouee: le 'exit status' est different [..
end
```

Exemple : Un cas de test qui réussit pour `lister`

(doit s'exécuter en signalant une erreur)

Cas de test du fichier `Tests/lister_spec.rb` générant une erreur


```
it_ "[...] erreur si fichier inexistant", :intermediaire do
  FileUtils.rm_f $DEPOT_DEFAUT

  genere_erreur /fichier.*#{$DEPOT_DEFAUT}.*existe pas/i do
    run_gv( 'lister' )
  end
end
```

Méthode `genere_erreur`

Fichier `Tests/test_helper.rb`


```
genere_erreur /fichier.*#{$DEPOT_DEFAUT}.*existe pas/i do
  .
  .
  .
end
```

1. Exécute le code dans le bloc, qui retourne un triplet :
   i. Tableau des lignes émises sur `stdout` (sans «`\n`»)
   ii. Tableau des lignes émises sur `stderr` (sans «`\n`»)
   iii. Code de statut
2. Vérifie que le 1^{er} résultat (lignes de `stdout`) est vide
3. Vérifie que le 2^{e} résultat (lignes de `stderr`) matche le motif décrivant l'erreur (en ignorant la casse : «`/.../i`»)
4. Vérifie que le code de statut n'est pas 0

Exemple : Un cas de test qui échoue pour `lister`

(devrait s'exécuter sans erreur)

Cas de test du fichier `Tests/lister_spec.rb`

Vérification stricte, car on teste l'option `--format`

```
it_ "inclut les items de la chaine", :intermediaire do
  avec_fichier $DEPOT_DEFAUT, lignes do
    genere_sortie ['Note pour 1 => 4',
                   'Note pour 5 => 3'], :strict do
      run_gv( 'selectionner --bus',
              '- lister --format="Note pour %I => %n"' )
    end
  end
end
```

Hypothèse pour l'exemple : Supposons qu'il y a une erreur dans `gv.sh` = le format `%n` est supposé utiliser la note associée au vin, mais à la place c'est le nom qui est retourné ☹

Cas de test du fichier `Tests/lister_spec.rb`

On voit ce qui est attendu (avec «−») vs. ce qui a été obtenu (avec «+»)

```
$ make test_lister NIVEAU=intermediaire
Run options: --seed 64285
# Running:
........F.
Finished in 1.437873s, 6.9547 runs/s, 30.6008 assertions/s.

  1) Failure:
GestionVins::lister::utilisation de format\
                    #test_0004_inclut les items de la chaine
[[...]/GestionVins/Tests/test_helper.rb:138]:
*** Assertion echouee: La sortie emise sur stdout n'est pas\
    *exactement* celle attendue (:strict => [...]) ***.
--- expected
+++ actual
@@ -1 +1 @@
-["Note pour 1 => 4", "Note pour 5 => 3"]
+["Note pour 1 => Volpaia", "Note pour 5 => Roseline"]


10 runs, 44 assertions, 1 failures, 0 errors, 0 skips
```

Exemple : Un cas de test qui échoue pour `lister`

(devrait s'exécuter en signalant une erreur)

Cas de test du fichier `Tests/lister_spec.rb`

Le message d'erreur émis n'est pas celui attendu

```
it_ "genere une erreur si arguments en trop", :intermediaire d
  lignes = IO.readlines("Tests/4vins.txt")
  attendu = [...]

  avec_fichier $DEPOT_DEFAUT, lignes do
    genere_erreur /Argument.*en trop/i do
      run_gv( 'lister foo' )
    end
  end
end
```

Hypothèse pour l'exemple : Dans `gv.sh`, une erreur est signalée et un message est bien émis, mais le message émis ne correspond pas à ce qui est attendu par le test ☹

Cas de test du fichier `Tests/lister_spec.rb`

Le message attendu est spécifié par une expression régulière («`/.../i`»), alors que la chaine indique le message effectivement émis

```
$ make test_lister NIVEAU=intermediaire
Run options: --seed 19424
# Running:
.F........
Finished in 1.410995s, 7.0872 runs/s, 32.6011 assertions/s.

  1) Failure:
GestionVins::lister\
  #test_0003_genere une erreur si arguments en trop
[[...]/GestionVins/Tests/test_helper.rb:119]:
*** Assertion echouee: Le message d'erreur[...]pas[...]attendu
Expected /Argument.*en trop/i to match\
        "*** Erreur: Il y trop d'argument: 'foo'".

10 runs, 46 assertions, 1 failures, 0 errors, 0 skips
make: *** [test_lister] Error 1
```

Donc : Le message d'erreur qui est émis n'est pas le bon ☹

⇒ Il faut donc le modifier.

Exemple : Un cas de test qui échoue pour `supprimer`

(devrait s'exécuter en retournant un code de statut non nul)

Cas de test du fichier `Tests/supprimer_spec.rb`

Un message d'erreur est émis et est correct, mais le code de statut est nul

```
it_ "genere une erreur si argument en trop", :intermediaire do
  avec_fichier $DEPOT_DEFAUT, lignes do
    genere_erreur /Nombre incorrect.*arguments?/i do
      run_gv( 'supprimer 4 foo' )
    end
  end
end
```

Hypothèse pour l'exemple : Dans `gv.sh`, un message d'erreur approprié est émis sur `stderr`, mais le code de statut retourné est nul ☹

Cas de test du fichier `Tests/supprimer_spec.rb`

```
$ make test_supprimer NIVEAU=intermediaire
Run options: --seed 7839
# Running:
.F...
Finished in 0.087365s, 57.2309 runs/s, 286.1544 assertions/s.

  1) Failure:
GestionVins::supprimer::cave avec plusieurs vins\
             #test_0003_genere une erreur si argument en trop
[[...]/GestionVins/Tests/test_helper.rb:120]:
*** Assertion echouee: le 'exit status' est 0 mais\
              devrait etre different de 0 ***.
Expected 0 to not be equal to 0.

5 runs, 25 assertions, 1 failures, 0 errors, 0 skips
make: *** [test_supprimer] Error 1
```

Donc : Il faut s'assurer qu'un code de statut non nul soit retourné en cas d'erreur $\Rightarrow$ Utilisez la fonction `erreur` !

Exemple : Un cas de test qui réussit pour `ajouter`

(ne doit rien émettre mais doit modifier le dépôt)

Autre méthode utilisée dans les tests fournis

Fichier `Tests/test_helper.rb`

- `execute_sans_sortie_ou_erreur` :
  1. Vérifie que rien n'a été émis sur `stdout`
  2. Vérifie que rien n'a été émis sur `stderr`
  3. Vérifie que le code de statut est 0

Cas de test du fichier `Tests/ajouter_spec.rb`

La méthode `avec_fichier` retourne, avec l'argument `:conserver`, le tableau des lignes du fichier après l'exécution

```
date = %x{date "+%d/%m/%y"}.chomp
```

```
it_ "ajoute dans un fichier vide" do
  nouveau_contenu = avec_fichier $DEPOT_DEFAUT,
                                 [],
                                 :conserver do
    execute_sans_sortie_ou_erreur do
      run_gv( 'ajouter Beaujolais 2017 Foillard 22.00' )
    end
  end

  # On s'assure que le fichier resultant a le bon contenu.
  nouveau_contenu.size.must_equal
    1
  nouveau_contenu.first.must_equal
    "1:#{date}:rouge:Beaujolais:2017:Foillard:22.00::"
end
```