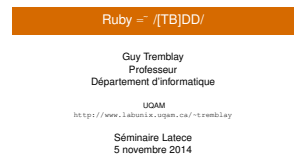
Ruby =~ /[TB]DD/

Guy Tremblay
Professeur
Département d'informatique

UQAM
http://www.labunix.uqam.ca/~tremblay

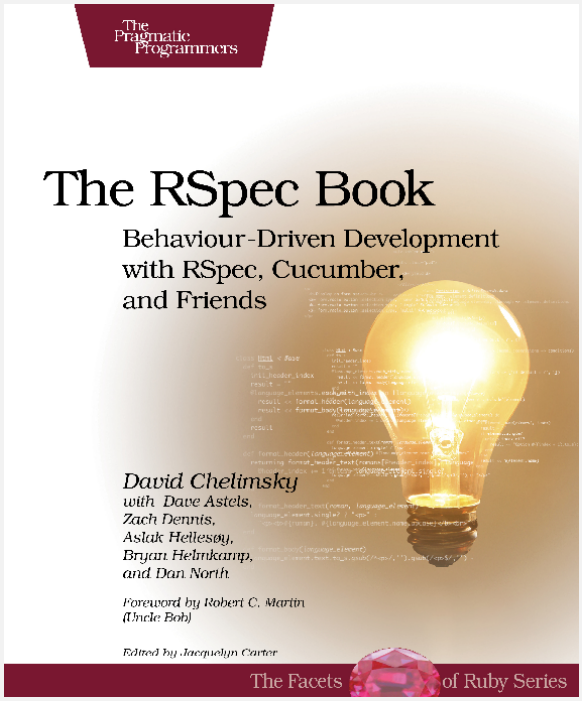
Séminaire Latece
5 novembre 2014

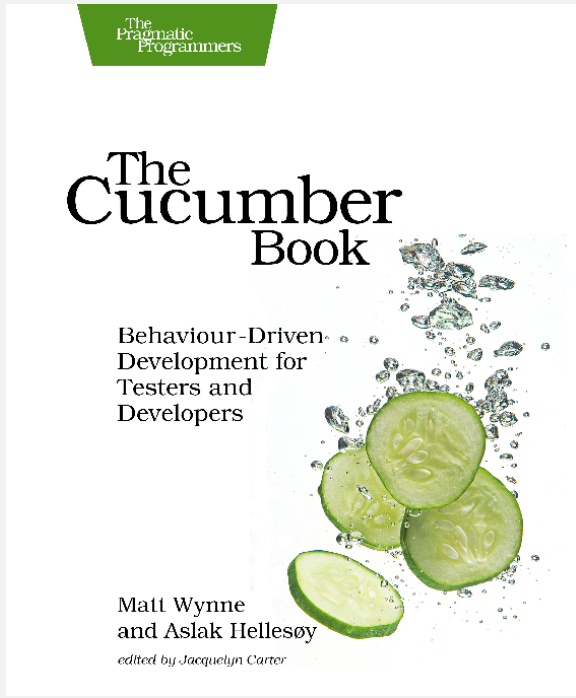
Ruby =~ /[TB]DD/

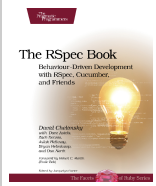


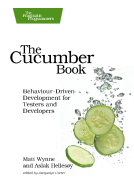
- Ce séminaire va traiter d'approches de développement de logiciels — TDD et BDD — et va le faire dans le contexte de «technologies» Ruby.
- On va donc voir dans quelle mesure Ruby est un bon match pour /[TB]DD/ !

Sources d’inspiration de ce séminaire



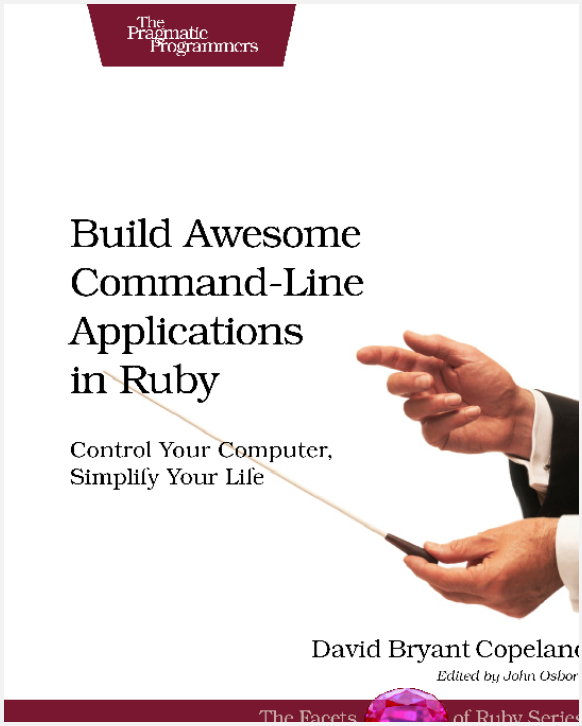


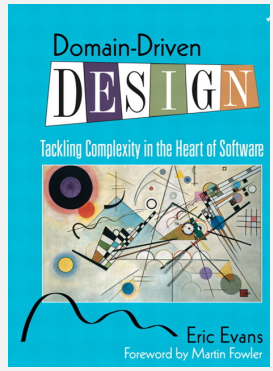




- Sources d’inspiration de ce séminaire = Des lectures intéressantes, que j’ai faites ces dernières années. J’avais lu la plupart de ces livres à l’époque où ils étaient sortis (il y a 2–3 ans), mais je n’avais pas eu le temps de vraiment bien les digérer et, surtout, les mettre en pratique car à l’époque j’étais directeur du département.
- Depuis le 1er août, je suis en mon congé sabbatique, j’en ai donc profité pour relire ces bouquins et, surtout, pour utiliser ces outils pour développer une petite application.

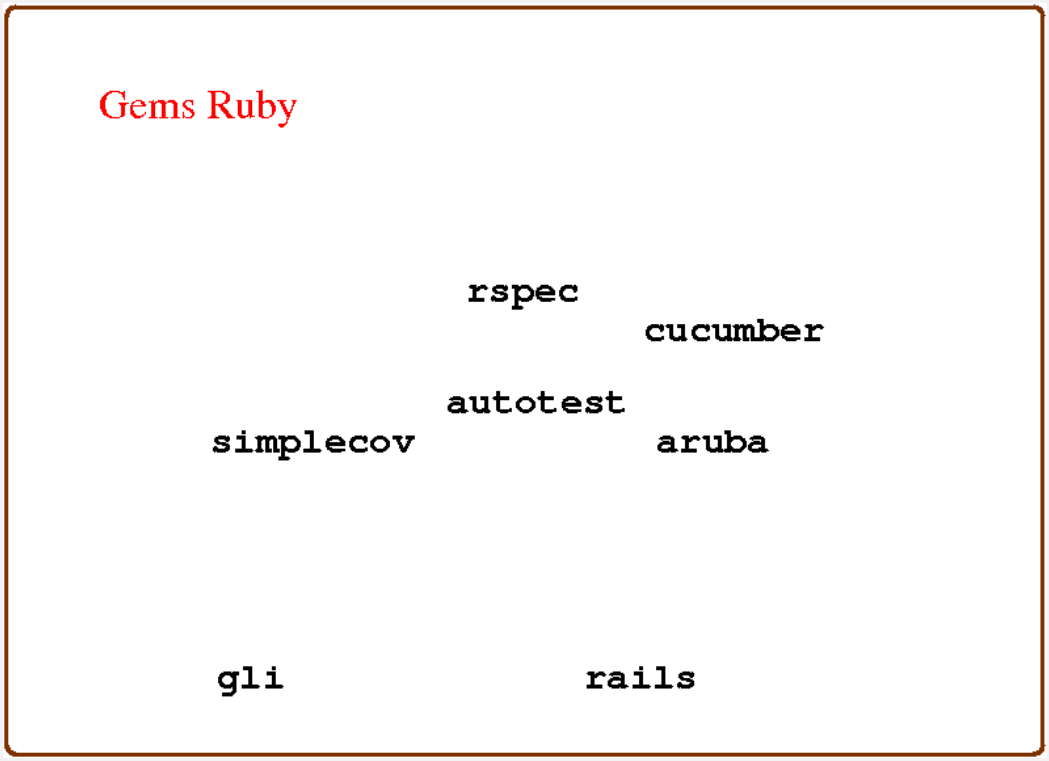
Sources d’inspiration de ce séminaire

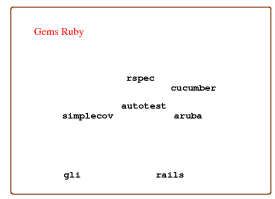




- Les trois principaux livres sont des livres sur Ruby, publiés dans «*The Facets of Ruby Series*» de la maison d’édition *The Pragmatic Bookshelf*, la maison d’édition d’où proviennent plusieurs des livres que j’ai achetés ces dernières années.
- L’autre bouquin, «*Domain Design Driven*», a aussi inspiré la façon dont j’ai développé l’application que je vais vous présenter. Par contre, faute de temps, je vous en parlerai très peu.

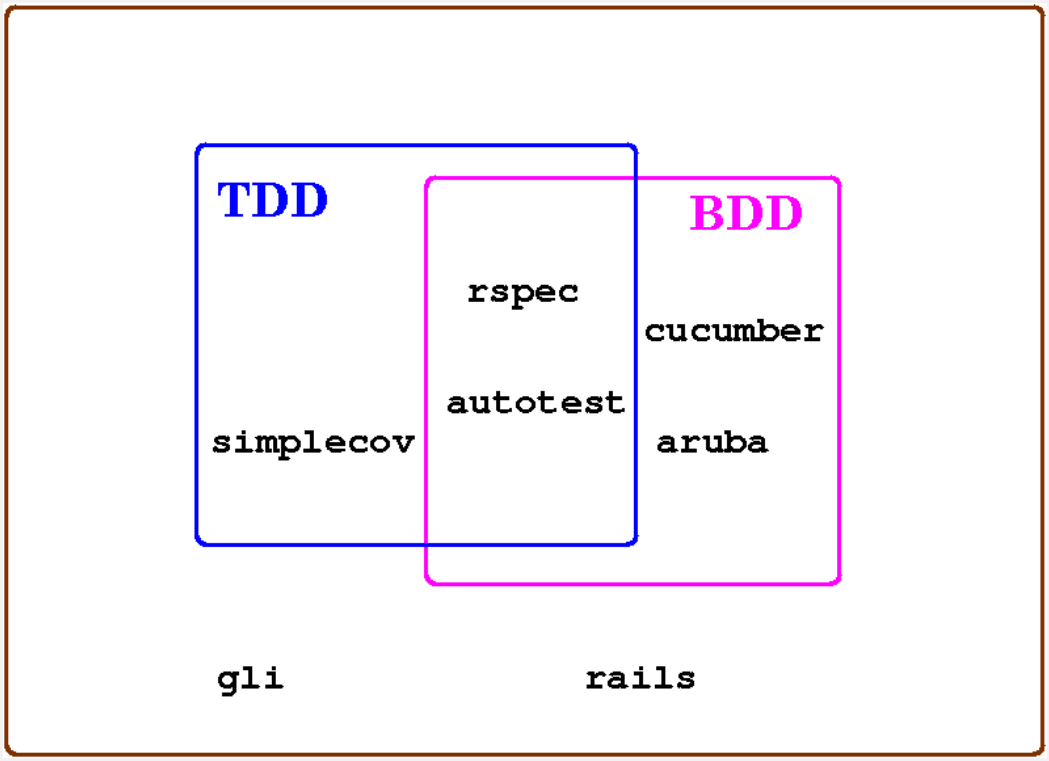
Thèmes abordés dans la présentation

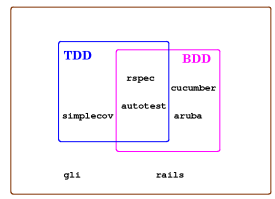




• Donc ces trois premiers bouquins, bien qu'ils présentent des outils Ruby, présentent quand même des approches plus générales, applicables dans des contextes autres que Ruby. Dans ce séminaire, je vais donc tenter de vous donner une idée générale de ces approches, mais en les illustrant avec des exemples Ruby, car depuis quelques années c'est assurément mon langage préféré.

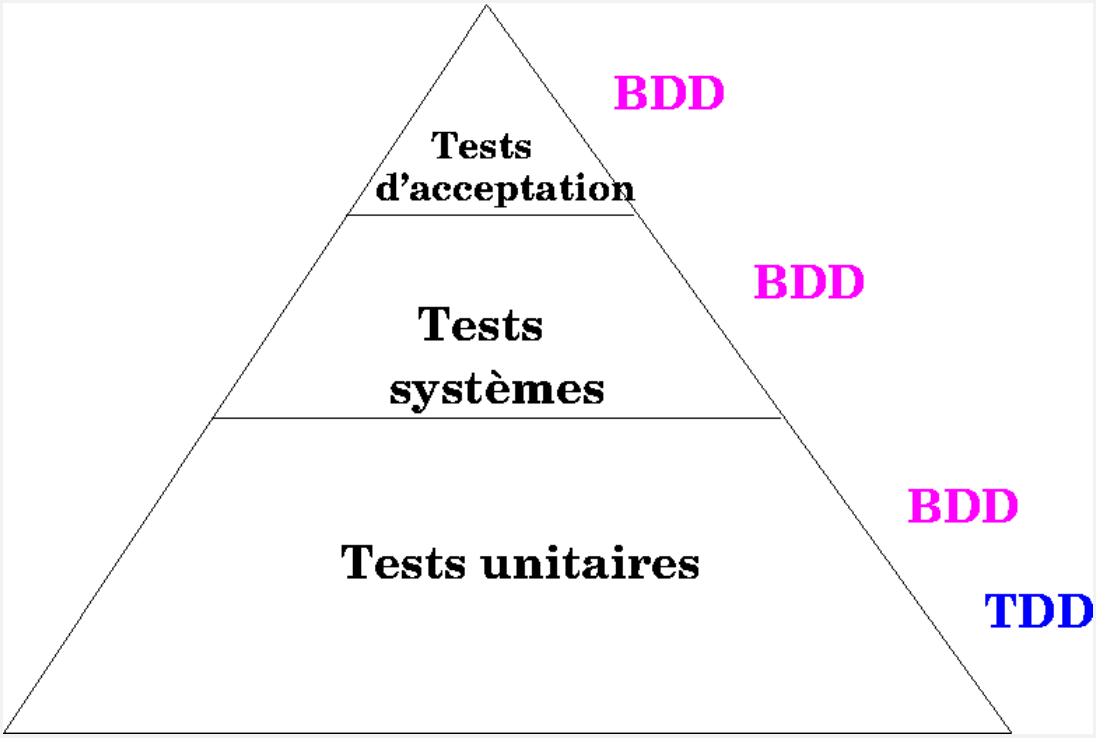
Thèmes abordés dans la présentation

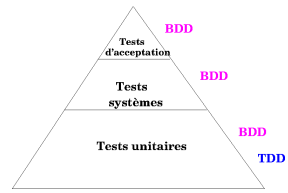




• Donc ces trois premiers bouquins, bien qu'ils présentent des outils Ruby, présentent quand même des approches plus générales, applicables dans des contextes autres que Ruby. Dans ce séminaire, je vais donc tenter de vous donner une idée générale de ces approches, mais en les illustrant avec des exemples Ruby, car depuis quelques années c'est assurément mon langage préféré.

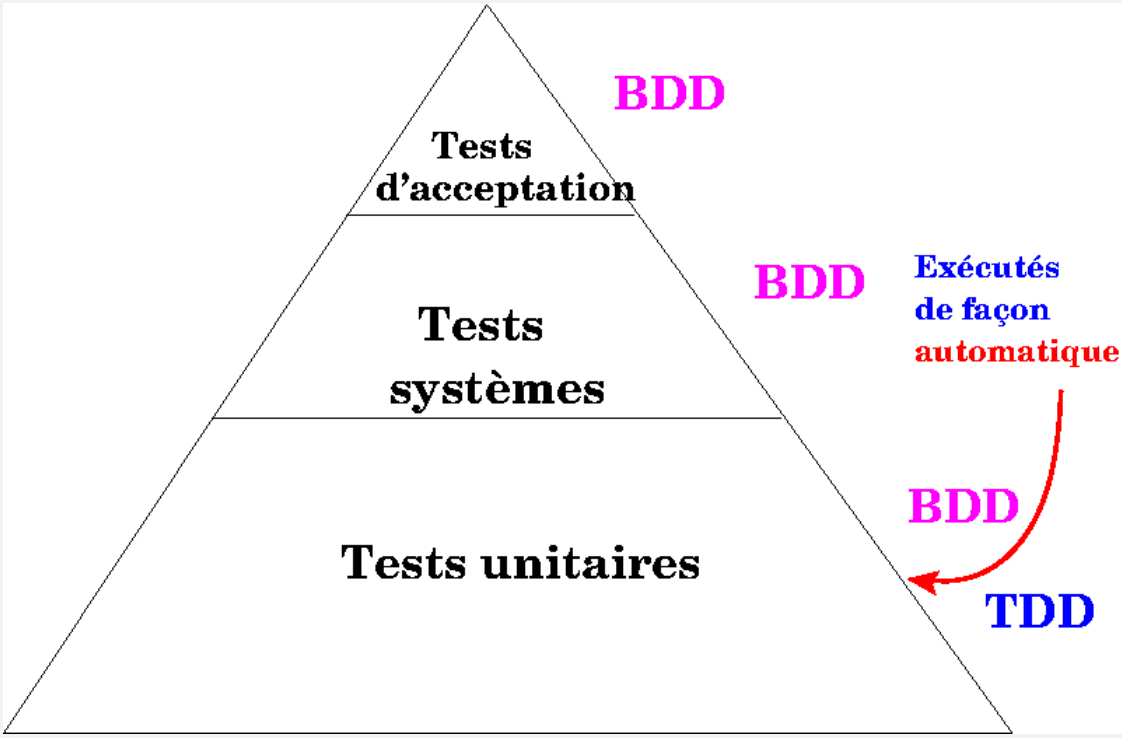
TDD et BDD s'appliquent à différents niveaux de tests

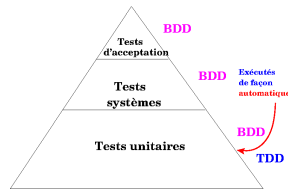




- Cela fait **très !** longtemps qu'on «dit» que les tests sont importants, qu'ils jouent un rôle clé dans le développement de logiciels de qualité.
- Ainsi, dans les vieux bouquins de Génie Logiciel, on trouve cette pyramide qui illustre les différents types de tests — leur hiérarchie.
- À la base, on a les tests unitaires, qui vérifient le bon fonctionnement *d'un module*, *composant*, *classe*, etc.
- Depuis longtemps on dit que les tests unitaires — sous la responsabilité des développeurs alors que les autres peuvent aussi impliquer testeurs, analystes, clients, etc. — sont importants.
- On voit aussi souvent, juste par dessus, «Tests d'intégration». Mais avec les outils et approches modernes, ces tests d'intégration sont aussi sous la responsabilité des développeurs, et les traite souvent comme une forme de tests «unitaires», où les unités sont de plus grande taille ou complexité — pas juste une classe mais un groupe de classes ou de composants.
- Dans ce qui suit, on va voir que ces différents niveaux

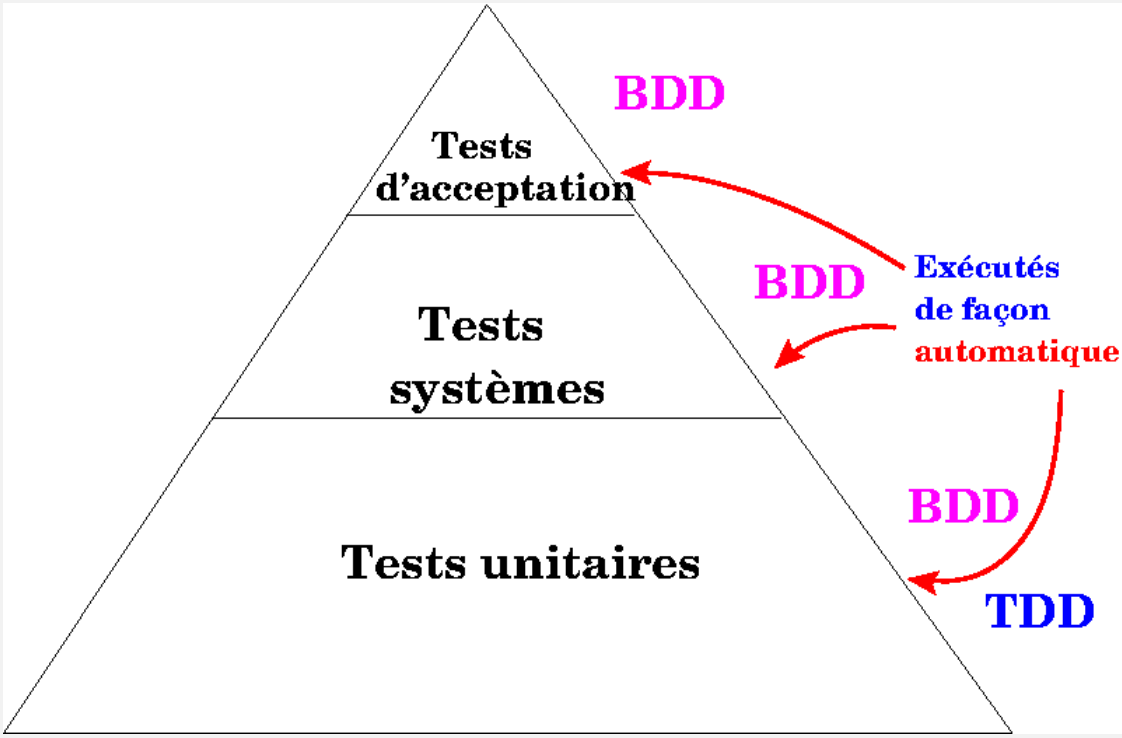
TDD et BDD s'appliquent à différents niveaux de tests





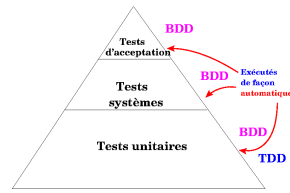
- Une caractéristique importante des tests unitaires c'est qu'ils doivent être exécutées souvent et de façon automatique, pous assurer la «non régression» du code.
- Sauf que ça ne fait pas si longtemps qu'on a des outils appropriés pour développer de tels tests. En fait, ce n'est qu'avec l'avènement des cadres de tests modernes — `JUnit` ! — qu'on a enfin eu des outils facilitant la spécification et l'exécution automatique des tests.
- Et ce qu'on va aussi voir, c'est qu'il n'y a pas de raison que ce soient uniquement les tests unitaires qui soient exécutées de façon automatique !

TDD et BDD s'appliquent à différents niveaux de tests



Ruby =~ /[TB]DD/

└TDD et BDD s'appliquent à différents niveaux de tests



-

Plan de la présentation

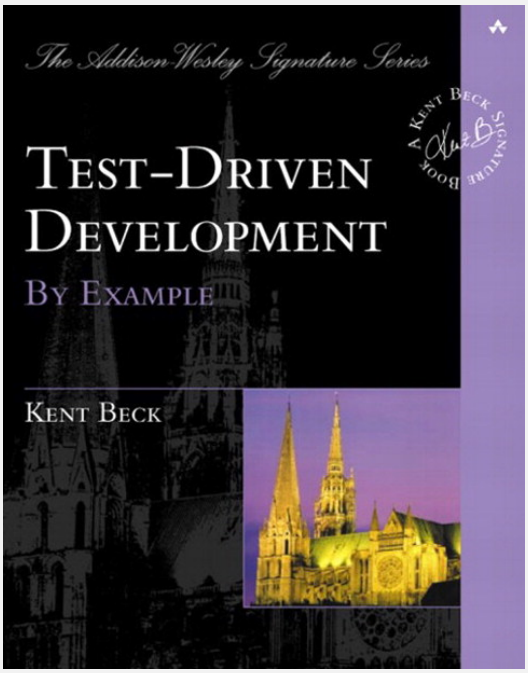
- TDD = *Test Driven Development*
- De TDD à BDD
  `RSpec`
- BDD = *Behavior Driven Development*
  `Cucumber`
- Un exemple : `biblio`

- TDD = *Test Driven Development*
- De TDD à BDD
  `RSpec`
- BDD = *Behavior Driven Development*
  `Cucumber`
- Un exemple : `biblio`

-

TDD = *Test Driven Development*

TDD

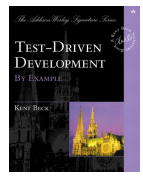


= Développement piloté par les tests

2016-11-04

Ruby =~ /[TB]DD/
└TDD = *Test Driven Development*

└TDD



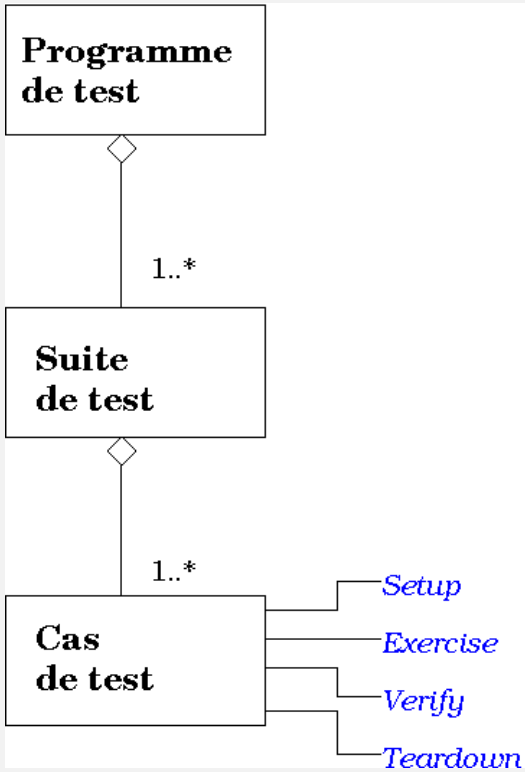
= Développement piloté par les tests

- L'approche TDD — *Test Driven Development* — a été initalement introduite en XP — *eXtreme Programming* — puis présentée à l'extérieur du contexte de XP par Beck, qui était aussi un des concepteurs de la méthode XP, dans un bouquin publié en 2003.
- Pour comprendre ce qu'est l'approche TDD, il faut tout d'abord bien comprendre ce que sont les tests unitaires et les cadres de tests (*test framework*, parce que l'exécution automatique et fréquente des tests est crucial pour cette approche.

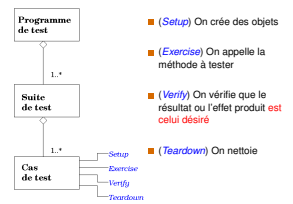
Pourquoi les tests sont-ils importants ?

• C’est certain qu’une des premières raison d’avoir des tests c’est d’assurer que le logiciel fonctionne correctement, qu’il n’y a pas de bogues.

Organisation typique d'un programme de test



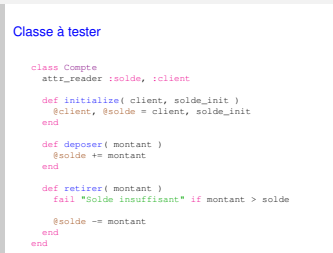
- (*Setup*) On crée des objets
- (*Exercise*) On appelle la méthode à tester
- (*Verify*) On vérifie que le résultat ou l'effet produit est celui désiré
- (*Teardown*) On nettoie



- Ce qui m'intéresse aujourd'hui, c'est l'aspect «exécution automatique».
- Pour comprendre l'aspect «exécution automatique», il faut comprendre la forme générale typique d'un programme et cas de test.
- Programme de test ⇒ Une ou plusieurs suites de test
- Suite de test ⇒ Un ou plusieurs cas de test
- Cas de test ⇒ «Une» (ou plusieurs) assertion(s)
- La vérification se fait à l'aide d'assertions

Exemple d'exécution automatique de tests : Classe à tester

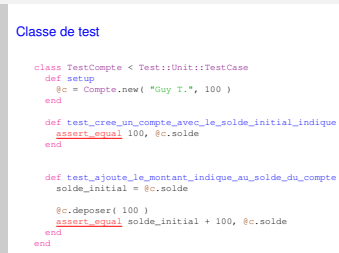
```ruby
class Compte
  attr_reader :solde, :client

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

  def deposer( montant )
    @solde += montant
  end

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

    @solde -= montant
  end
end
```

- On va illustrer cela avec un petit exemple en Ruby et le cadre de tests `Test::Unit`.
- Soit une classe pour un `Compte` bancaire — ultra simplifiée !
- Méthode `initialize` : Appelée lors de l'allocation d'un objet avec `Compte.new`.

Exemple d'exécution automatique de tests : Classe de test

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

  def test_cree_un_compte_avec_le_solde_initial_indique
    assert_equal 100, @c.solde
  end


  def test_ajoute_le_montant_indique_au_solde_du_compte
    solde_initial = @c.solde

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

- Voici un petit programme de tests : une classe `TestCompte` hérite de la classe `Test::Unit::TestCase`. Les méthodes de `TestCompte` qui débutent par `test_` sont les méthodes de test — on dit aussi les «cas de test».
- Les méthodes de test utilisent diverses formes «*d'assertions*» pour vérifier les résultats. Une assertion ne produit aucun résultat (ou presque) si la condition de l'assertion est vraie.
- La méthode `setup` est appelée avant l'exécution de chacune des méthodes de test.

Exemple d'exécution automatique de tests : Résultat d'exécution sans erreur

```
$ ruby test-compte.rb
Loaded suite test-compte
Started
..

Finished in 0.001382 seconds.

2 tests, 2 assertions, 0 failures, 0 errors, 0 pendings,
0 omissions, 0 notifications
100% passed

1447.18 tests/s, 1447.18 assertions/s
```


Résultat d'exécution sans erreur

$ ruby test-compte.rb
Loaded suite test-compte
Started
..

Finished in 0.001382 seconds.

2 tests, 2 assertions, 0 failures, 0 errors, 0 pendings,
0 omissions, 0 notifications
100% passed

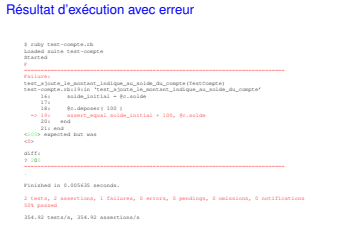
1447.18 tests/s, 1447.18 assertions/s

- L'exécution du programme de test produira alors, si tout est ok, les résultats suivant.

Exemple d'exécution automatique de tests : Résultat d'exécution avec erreur

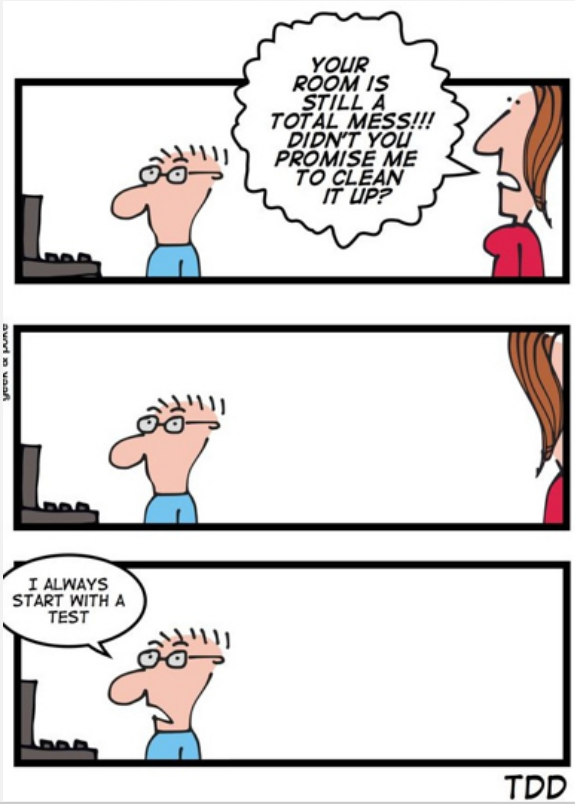
```
$ ruby test-compte.rb
Loaded suite test-compte
Started
F
===============================================================================
Failure:
test_ajoute_le_montant_indique_au_solde_du_compte(TestCompte)
test-compte.rb:19:in 'test_ajoute_le_montant_indique_au_solde_du_compte'
     16:     solde_initial = @c.solde
     17:
     18:     @c.deposer( 100 )
  => 19:     assert_equal solde_initial + 100, @c.solde
     20:   end
     21: end
<200> expected but was
<0>

diff:
? 200
===============================================================================
.

Finished in 0.005635 seconds.

2 tests, 2 assertions, 1 failures, 0 errors, 0 pendings, 0 omissions, 0 notifications
50% passed

354.92 tests/s, 354.92 assertions/s
```


Résultat d'exécution avec erreur



- Et voici un exemple d"exécution du programme de test avec des erreurs.

Qu’est-ce que le TDD ?

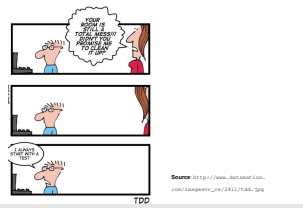


Source: `http://www.datamation.com/imagesvr_ce/2411/tdd.jpg`

2016-11-04

Ruby =~ /[TB]DD/
└TDD = *Test Driven Development*

└Qu’est-ce que le TDD ?



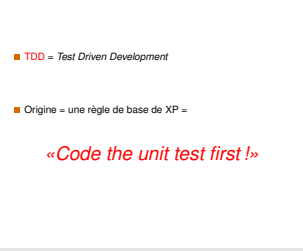
-

Qu’est-ce que le TDD ?

- TDD = *Test Driven Development*

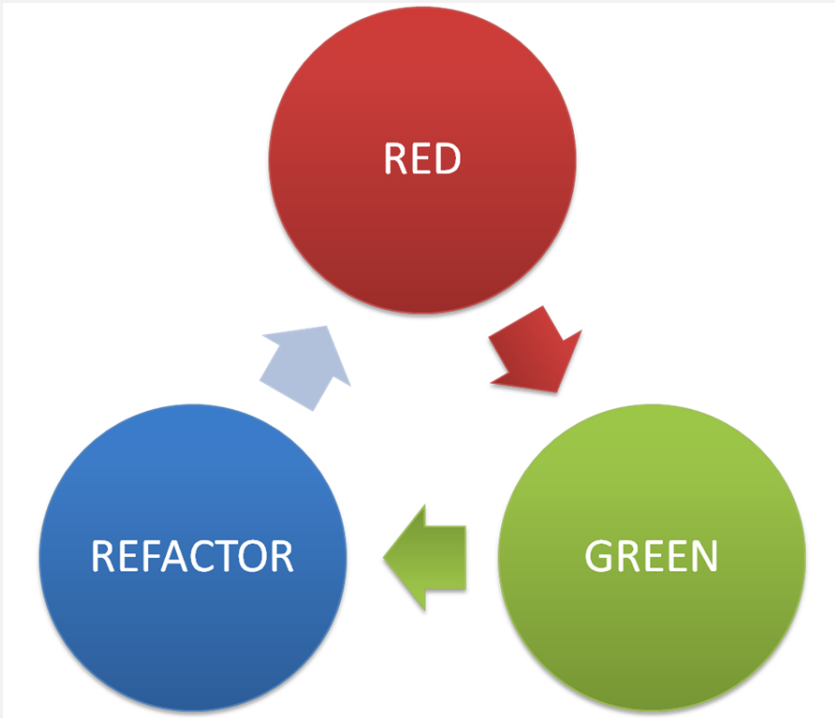
- Origine = une règle de base de XP =

«Code the unit test first !»

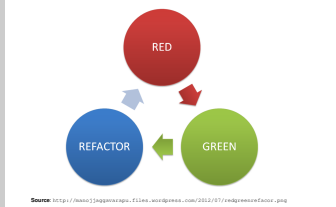


- XP = *eXtreme Programming*
- Donc, selon cette approche, on ne devrait jamais écrire une ligne de code tant qu’on a pas un test qui montre qu’on a besoin de ce code — parce que le test échoue (sic)

L’approche TDD «pure et dure»

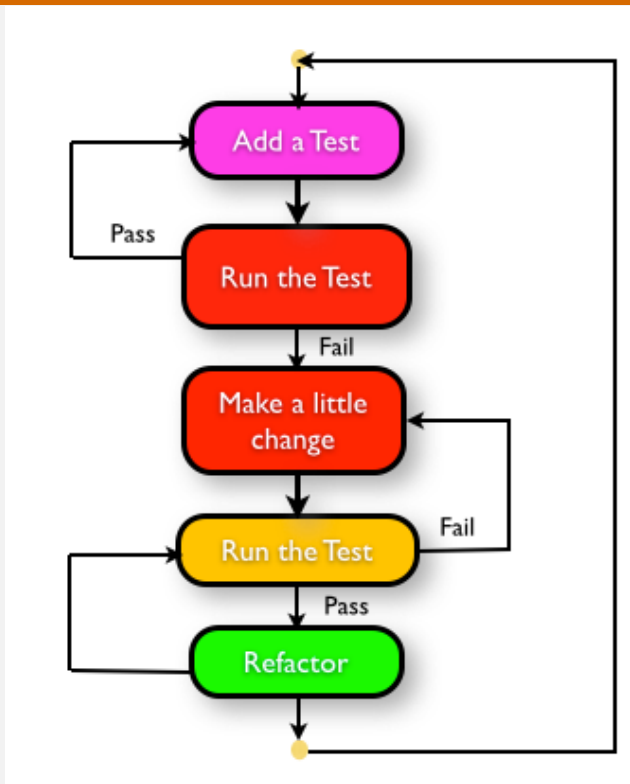


Source: `http://manojjaggavarapu.files.wordpress.com/2012/07/redgreenrefacor.png`

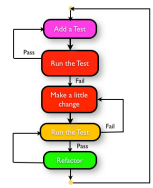


- Voici un diagramme qu’on voit souvent lors de la présentation de TDD.
- Je ne trouve pas que c’est la figure qui illustre le mieux l’approche et je vais donc vous en présenter une autre.
- Rouge : Un ou plusieurs tests ne passent pas, donc on doit écrire ou modifier du code.
- Vert : Tous tests passent, Donc c’est ok… et on peut alors modifier notre code avec l’assurance de ne pas régresser car les tests nous diront si on a changé le comportement — si on est retourné dans le rouge !
- Refactoriser = améliorer le code tout en gardant exactement les mêmes fonctionnalités

Approche TDD «pure et dure» (bis)



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



- Les cinq étapes de l'approche TDD «pure et dure» :
  1. On écrit un premier test ;
  2. On vérifie qu'il échoue (*) – et donc que le test est valide ;
  3. On écrit juste le code suffisant pour passer le test ;
  4. On vérifie que le test passe ;
  5. On refactorise le code.

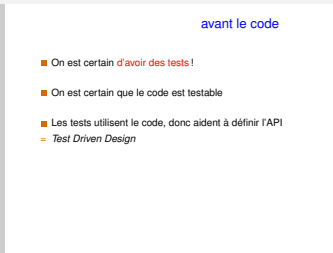
Source : `http://fr.wikipedia.org/wiki/Test_Driven_Development`

- Approche «pure et dure», pour ne pas dire, parfois, «dogmatique»

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*

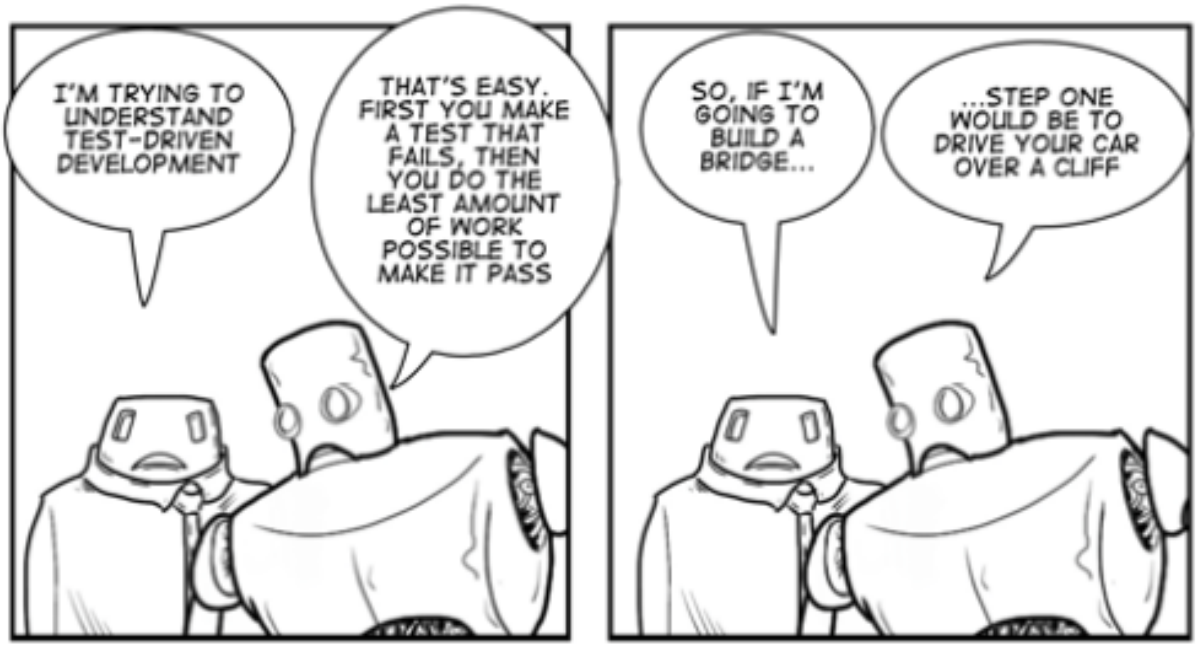


• Avant l'apparition des cadres de tests et l'exécution automatique des tests, on devait souvent exécuter manuellement les tests, ce qui était long et pénible. Donc, même si en théorie on disait que les tests étaient importants, souvent il n'y en avait pas. Mais la situation a clairement changé avec l'apparition des cadres de test.

• Par testable, on veut dire qu'on est capable *i* de définir un contexte approprié à l'appel d'une méthode, *ii* d'appeler la méthode, puis *iii* de vérifier que le résultat produit est bien celui attendu. On peut assurément imaginer des méthodes ou classes ou modules qui sont tellement mal foutues… qu'on arrive même pas à savoir si elles fonctionnent correctement.

• Test Driven Design = Conception dirigée par les tests : Beck mettait beaucoup l'accent là-dessus.

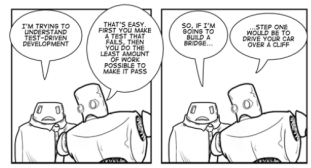
Un aspect controversé de TDD



2016-11-04

Ruby =~ /[TB]DD/
└TDD = *Test Driven Development*

└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`

[1m37 à 2m25]

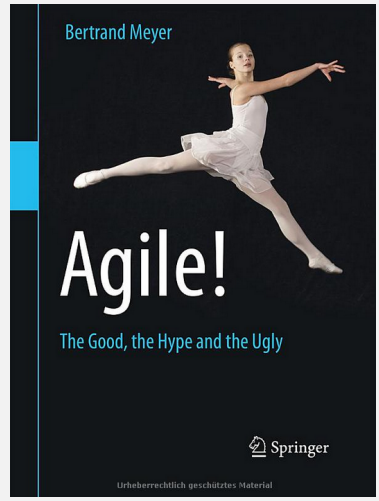
Les trois lois de TDD selon *Uncle Bob*

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**.»



Self-testing code !

2016-11-04

Ruby =~ /[TB]DD/
└TDD = *Test Driven Development*
└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 !

- Personnnellement, cela fait maintenant plusieurs années que je développe mes programmes «en même temps» que mes tests. En fait, en 1998, donc avant même l'apparition des cadres de tests comme JUnit, j'avais développé un paquet de *shell* scripts pour tester automatiquement un «compilateur».
- Par contre, j'ai rarement utilisé une approche «TDD pure et dure». Et quand j'écris un test *avant* d'écrire un bout de code, je n'essaye pas nécessairement d'écrire le bout de code le plus simple possible qui fait l'affaire, ce qui est supposé être la règle quand on fait du TDD «pur et dur». En fait, je trouve que certains des exemples de cette pratique. . . ne me semblent pas réaliste ☹
- Par contre, je ne fais jamais un déploiement tant que tous les tests n'ont pas été développés et qu'ils passent. Jamais de `push` public (sur `master`) tant qu'il n'y a pas des tests appropriés.
- Dixit Martin Fowler : «Self-Testing Code is the name I used in "*Refactoring*" to refer to the practice of writing comprehensive

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

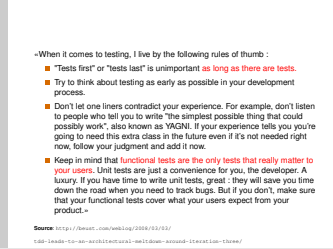
«When it comes to testing, I live by the following rules of thumb :

- "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/`

2016-11-04

Ruby =~ /[TB]DD/
└TDD = *Test Driven Development*
└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/

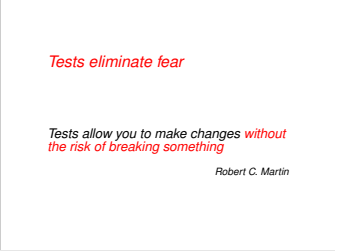
-

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

Tests eliminate fear

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

Robert C. Martin



- Donc, une fois qu'on a des tests pour notre code, on sait qu'on peut modifier notre code pour le nettoyer, l'améliorer, etc., sans danger de régresser—i.e., sans danger de retourner en arrière dans un état où le logiciel ne fonctionne plus.

La présence de tests permet/favorise le *refactoring*

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

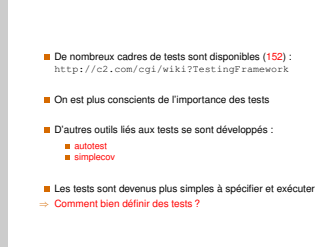
Ruby =~ /[TB]DD/
└TDD = *Test Driven Development*
└La présence de tests permet/favorise le *refactoring*

-

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

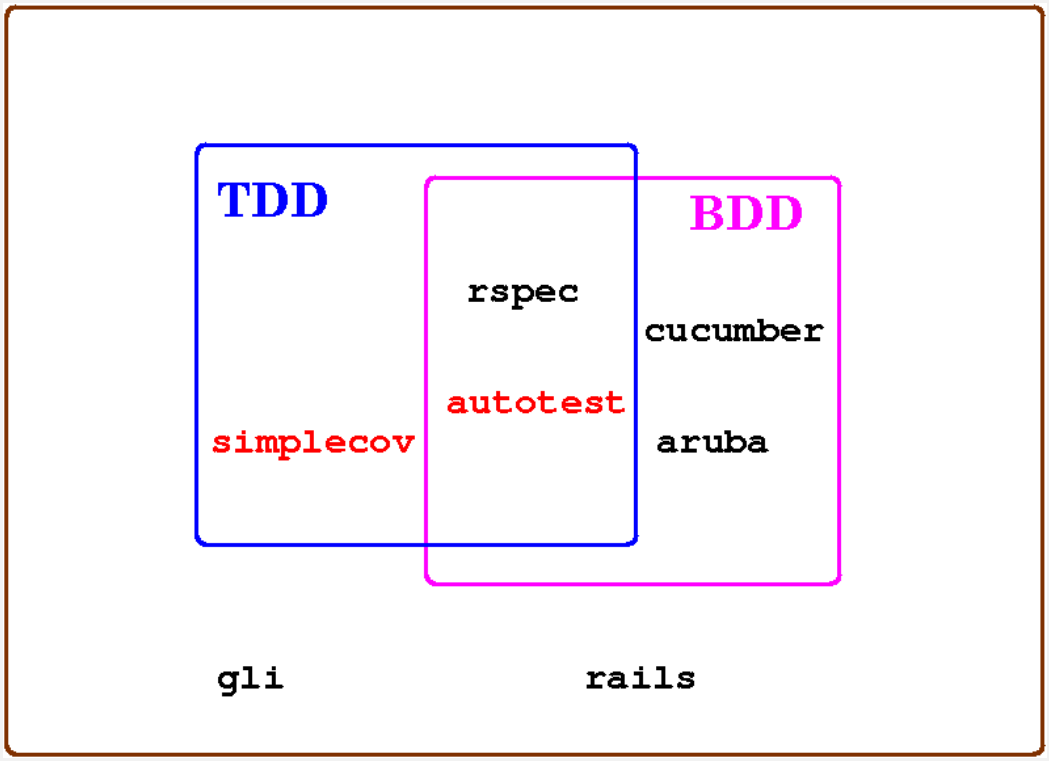
- De nombreux cadres de tests sont disponibles (152) : `http://c2.com/cgi/wiki?TestingFramework`
- On est plus conscients de l'importance des tests
- D'autres outils liés aux tests se sont développés :
  - autotest
  - simplecov
- Les tests sont devenus plus simples à spécifier et exécuter

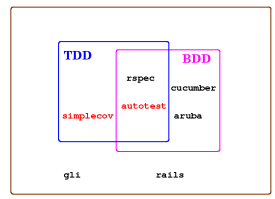
⇒ Comment bien définir des tests ?



• Un des effets marquants de JUnit a été le développement de nombreux cadres de tests, dans différents langages, inspirés de JUNit — on parlait alors de `xUnit` et il y avait un site dédié, que je n'ai pas retrouvé. Mais Cuningham en répertorie un très grand nombre sur son site.
• J'ai moi-même développé un tel cadre de tests. Pendant quelques années, nous avons utilisé le langage MPD dans les cours de programmation parallèle, et il n'y avait pas de cadre de tests pour ce langage, donc j'en ai développé un = `MPDUnit`.
• Mais ce dont je veux parler aujourd'hui, ce sont plutôt des cadres de tests qui utilisent une approche des cadres xUnit, qui vont aussi au-delà… des tests unitaires.
• Et par bien définir, je veux tout d'abord traiter l'aspect de comment «bien nommer les tests».

D’autres outils liés aux tests





- `autotest` : relance l’exécution des tests aussitôt qu’on modifie un fichier — sauvegarde avec l’éditeur de texte. Donc, permet de ne pas avoir à se casser la tête pour déterminer quand relancer les tests. Parfois «agaçant», mais parfois très utile car ça nous signale quand les tests ne passent plus… alors qu’on pensait que tout était ok.
- `autotest` : si on respecte certaines conventions, peut ne lancer que les tests des éléments ayant été modifiés.
- `simplecov` : couverture simple du code, pas une couverture sophistiquée, mais quand même utile.

De TDD à BDD

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, y compris en cas d'erreur

Le nom d'une méthode de test devrait être une phrase qui nous éclaire sur le comportement de la méthode testée, y compris en cas d'erreur

- Pour bien nommer les cas de test, il faut tenir compte que pour une méthode du code à tester, par exemple, `retirer`, on aura plusieurs cas de tests à écrire — plusieurs méthodes de tests — et ce pour tester différents aspects, différentes situations : retirer une partie, retirer tout, retirer plus que le solde, etc. On ne pourra pas simplement appeler ces méthodes `tester_retirer`, car ce ne serait pas assez spécifique.
- De plus, il faut aussi avoir des cas de test les plus indépendants possibles, pour bien cerner/identifier les erreurs problèmes possibles quand quelque chose ne fonctionne pas.
- Mais ceci a amené une autre question, à savoir : quel est un nom approprié, significatif, pour un cas de test (une méthode de test) ? Parce que les règles pour le nommage des méthodes habituelles ne s'appliquent pas, style, prédicat pour les observateur, verbe pour les actions ! Le nom de la méthode de test doit plutôt nous renseigner sur l'aspect du module/classe/méthode qui est testé.

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() );
}
```


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() );
}
```


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 : Suggestion

Convention suggérée pour les noms des méthodes de test :

NomDeMéthode_ÉtatTesté_RésultatAttendu

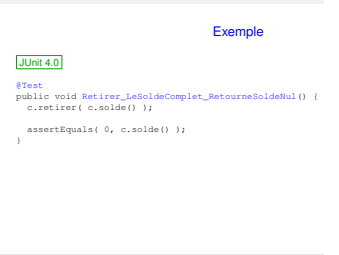
Suggestion
Convention suggérée pour les noms des méthodes de test :
NomDeMéthode_ÉtatTesté_RésultatAttendu

- Motivations de cette convention :
  - Le nom du test devrait exprimer une exigence spécifique
  - Le nom du test devrait inclure les données ou l'état, tant en entrée qu'en sortie
  - Le nom du test devrait inclure le nom de la méthode ou classe testée
- Autre motivation :
  - Le nom du test devrait nous aider à comprendre le comportement attendu y compris (surtout !) lorsque le test «échoue».

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() );
}
```

- Donc, en Java avec `JUnit`, ça peut donner un nom de méthode qui aurait l'allure suivante, donc pas nécessairement facile à lire, et encore moins à écrire ☹

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

Ruby (`Test::Unit`)

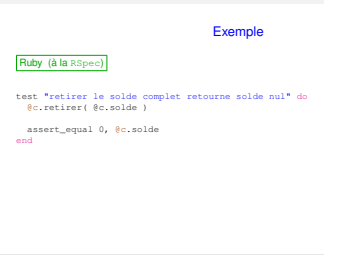
```ruby
def test_retirer_le_solde_complet_retourne_solde_nul
  @c.retirer( @c.solde )

  assert_equal 0, @c.solde
end
```


- En Ruby, avec `Test::Unit`, ça donnerait cela — sauf pour les noms de classe, on utilise pas le *CamelCase* en Ruby. Là non plus, ce n'est pas l'idéal.

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

Ruby (à la RSpec)

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

  assert_equal 0, @c.solde
end
```

- Par contre, en Ruby, il y a plein de «trucs» qu'on peut faire, difficile ou impossible à faire en Java.
- Par exemple, cette façon d'écrire le nom du test ne serait-elle pas plus simple à écrire et à lire ?

(Parenthèse : Définition Ruby d'une méthode `test`)

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

def test( ident, &bloc )
  define_method symbole_pour("test_", ident) do
    instance_eval &bloc
  end
end
```


(Parenthèse : Définition Ruby d'une méthode test)

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

def test( ident, &bloc )
  define_method symbole_pour("test_", ident) do
    instance_eval &bloc
  end
end
```

- On va faire une brève digression, pour vous montrer que c'est relativement facile, en Ruby, de passer de l'une à l'autre façon. Pour ce faire, il suffit de se définir une méthode… de quelques lignes, qui ressemble à ça !
- Par contre, dans ce qui suit, on va aussi voir… qu'une telle approche est celle supportée par le cadre de tests JUnit.

Suggestion de D. North (2003)

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

Exemple :

Retirer

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

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

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

- ...

JBehave (North, 2003)

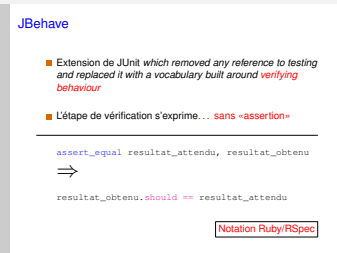
- Extension de JUnit *which removed any reference to testing and replaced it with a vocabulary built around verifying behaviour*
- L'étape de vérification s'exprime... sans «assertion»

```
assert_equal resultat_attendu, resultat_obtenu
```

$\Rightarrow$

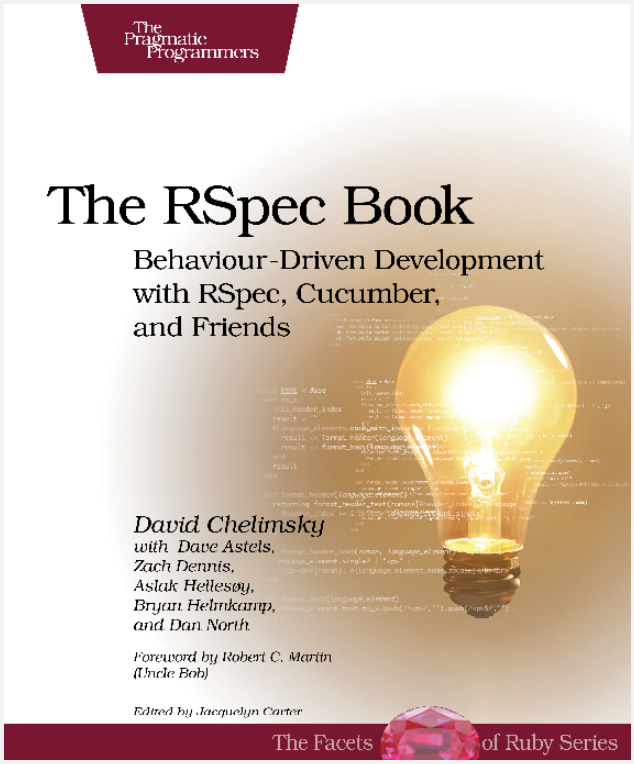
```
resultat_obtenu.should == resultat_attendu
```

Notation Ruby/RSpec

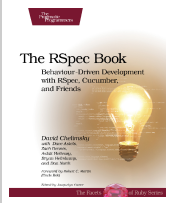


• Donc, North, en 2003, a développé JBehave, un outil permettant de spécifier des cas de tests en Java en s'inspirant de ces constatations et suggestions.
• Dans ce qui suit, je vais toutefois l'illustrer en Ruby, avec `RSpec`, avec lequel je suis plus familier... et qui a eu plus de succès que JBehave.

RSpec (Chelimsky, North et al., 2007)



Cadre de tests Ruby inspiré de JBehave



Cadre de tests Ruby inspiré de JBehave

-

Exemple RSpec : Définition de `Compte`

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

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

  def deposer( montant )
    @solde += montant
  end

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

    @solde -= montant
  end
end
```

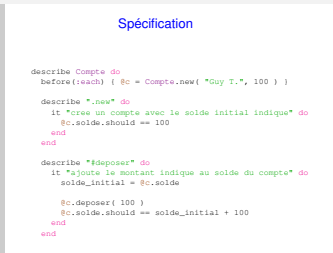

Définition

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

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

  def deposer( montant )
    @solde += montant
  end

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

    @solde -= montant
  end
end
```

- Donc, soit à nouveau notre classe pour un `Compte` bancaire simple.

Exemple RSpec : Spécification de `Compte` (1)

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

  describe ".new" do
    it "cree un compte avec le solde initial indique" do
      @c.solde.should == 100
    end
  end

  describe "#deposer" do
    it "ajoute le montant indique au solde du compte" do
      solde_initial = @c.solde

      @c.deposer( 100 )
      @c.solde.should == solde_initial + 100
    end
  end
```

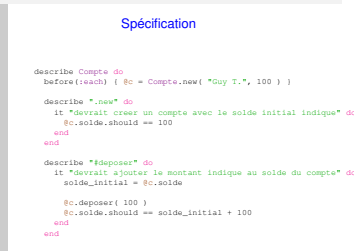
- Voici des tests équivalents à ceux vus précédemment, avec en plus des tests pour `retirer`.
- Éléments de «convention» pour décrire les tests :
  - Le premier niveau de `describe` indique pour quelle classe on est en train de spécifier le comportement.
  - Chaque niveau interne de `describe` indique alors la méthode décrite/spécifiée — avec le préfixe «`.`» pour les méthodes de classe et avec le préfixe «`#`» pour les méthodes d'instance.
  - Un `it` représente un cas de test.
- Les assertions utilisées sont exprimées d'une façon différente. Dans les premières versions de RSpec, on utilisait des `should` :

```ruby
assert_equal solde_initial+100, @c.solde
```

```ruby
@c.solde.should == solde_initial + 100
```

Exemple RSpec : Spécification de `Compte` (1)

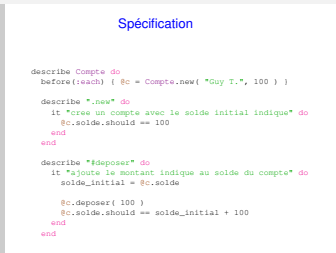
```
describe Compte do
  before(:each) { @c = Compte.new( "Guy T.", 100 ) }

  describe ".new" do
    it "devrait creer un compte avec le solde initial indique" do
      @c.solde.should == 100
    end
  end

  describe "#deposer" do
    it "devrait ajouter le montant indique au solde du compte" do
      solde_initial = @c.solde

      @c.deposer( 100 )
      @c.solde.should == solde_initial + 100
    end
  end
```

- Certains suggéraient, les premiers temps, que les tests débutent avec «should». Mais maintenant, c'est considéré comme une mauvaise pratique : pourquoi dire «devrait faire X» quand on peut dire tout aussi clairement et simplement «fait X».

Exemple RSpec : Spécification de `Compte` (1)

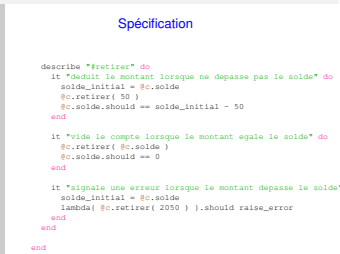
```ruby
describe Compte do
  before(:each) { @c = Compte.new( "Guy T.", 100 ) }

  describe ".new" do
    it "cree un compte avec le solde initial indique" do
      @c.solde.should == 100
    end
  end

  describe "#deposer" do
    it "ajoute le montant indique au solde du compte" do
      solde_initial = @c.solde

      @c.deposer( 100 )
      @c.solde.should == solde_initial + 100
    end
  end
```

-

Exemple RSpec : Spécification de `Compte` (2)

```
  describe "#retirer" do
    it "deduit le montant lorsque ne depasse pas le solde" do
      solde_initial = @c.solde
      @c.retirer( 50 )
      @c.solde.should == solde_initial - 50
    end

    it "vide le compte lorsque le montant egale le solde" do
      @c.retirer( @c.solde )
      @c.solde.should == 0
    end

    it "signale une erreur lorsque le montant depasse le solde" do
      solde_initial = @c.solde
      lambda{ @c.retirer( 2050 ) }.should raise_error
    end
  end

end
```

- Cet exemple pour `retirer` illustre qu'il peut y avoir , et c'est généralement le cas, plusieurs cas de tests (plusieurs `it`) pour une méthode à tester.
- Dans les versions plus récentes de RSpec, on n'utilise même plus des `should` dans les assertions, on utilise plutôt des `expect` — parce que les `should` créaient parfois des problèmes dans certains programmes — mise en oeuvre avec *«Monkey patching»*.
- Les avis sont assez partagés quant à cette nouvelle forme. Personnellement, je préfère nettement l'ancienne forme, c'est cela qui m'a attiré vers RSpec. L'autre forme ressemble plus aux anciennes assertions, même si c'est quand même différent.

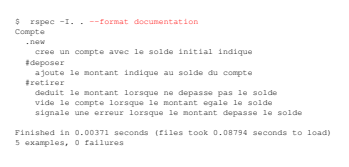
Résultats d'exécution : Format `progress` (défaut)

```
$ rspec -I. .
.....

Finished in 0.00361 seconds (files took 0.08863 seconds to load)
5 examples, 0 failures
```


- Format `progress` = format par défaut = sortie style JUnit standard
- On remarque qu'on ne parle plus de tests mais d'exemples.

Résultats d'exécution : Format `documentation`

```
$  rspec -I. . --format documentation
Compte
  .new
    cree un compte avec le solde initial indique
  #deposer
    ajoute le montant indique au solde du compte
  #retirer
    deduit le montant lorsque ne depasse pas le solde
    vide le compte lorsque le montant egale le solde
    signale une erreur lorsque le montant depasse le solde

Finished in 0.00371 seconds (files took 0.08794 seconds to load)
5 examples, 0 failures
```

- Format `documentation` = donne les différents tests exécutés avec les `describe`s et les `it`s !
- Si les phrases de `describe` et des `it` sont bien formulées, cela peut parfois/souvent ressembler à une spécification informelle !
- Donc, les résulats attendus de l'exécution des tests deviennent une spécification du comportement de l'entité.

BDD = *Behavior Driven Development*

Constatation de North et Matts (2004) : L'approche BDD peut aussi s'appliquer... aux exigences

Toward the end of 2004, while I was describing my new found, behaviour-based vocabulary to Matts, he said, "But that's just like analysis." There was a long pause while we processed this, and then we decided to apply all of this behaviour-driven thinking to defining requirements.

Source: *Dan North,*
`http://dannorth.net/introducing-bdd/`

Toward the end of 2004, while I was describing my new found, behaviour-based vocabulary to Matts, he said, "But that's just like analysis." There was a long pause while we processed this, and then we decided to apply all of this behaviour-driven thinking to defining requirements.

Source: *Dan North,*
`http://dannorth.net/introducing-bdd/`

Constatation de North et Matts (2004) : L'approche BDD peut aussi s'appliquer. . . aux exigences

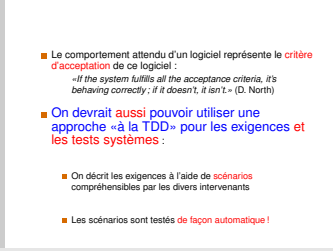
- Le comportement attendu d'un logiciel représente le critère d'acceptation de ce logiciel :

  «If the system fulfills all the acceptance criteria, it's behaving correctly ; if it doesn't, it isn't.» (D. North)

- On devrait aussi pouvoir utiliser une approche «à la TDD» pour les exigences et les tests systèmes :

  - On décrit les exigences à l'aide de scénarios compréhensibles par les divers intervenants

  - Les scénarios sont testés de façon automatique !



- Donc, dans un premier temps, BDD est une façon de spécifier des tests unitaires en utilisant un style différent de spécification — spécification par des exemples.
- Donc on devrait pouvoir faire du TDD aussi à l'étape d'analyse
- Avec des scénarios qui soient exécutables, comme des tests
- Qu'on pourra tester lors des tests systèmes et d'acceptation

Qu’est-ce que le BDD ?

«[Le] BDD consiste à étendre le TDD en écrivant non plus du code compréhensible uniquement par des développeurs, mais sous forme de scénario compréhensible par toutes les personnes impliquées dans le projet.

Autrement dit, il s’agit d’écrire des tests qui décrivent le comportement attendu du système et que tout le monde peut comprendre.»

Source: `http://arnauld.github.io/incubation/Getting-Started-with-JBehave.html`

«[Le] BDD consiste à étendre le TDD en écrivant non plus du code compréhensible uniquement par des développeurs, mais sous forme de scénario compréhensible par toutes les personnes impliquées dans le projet.

Autrement dit, il s’agit d’écrire des tests qui décrivent le comportement attendu du système et que tout le monde peut comprendre.»

Source: `http://arnauld.github.io/incubation/Getting-Started-with-JBehave.html`

- «[G]énéralement, ces scénarios sont écrits et définis avant que l’implémentation ne commence. Ils servent à la fois à définir le besoin mais vont guider le développement en le focalisant sur la fonctionnalité décrite. Dans l’absolu, on continue à faire du TDD mais on ajoute en plus l’expression du besoin en langage naturel. Alors que le TDD garantit d’une certaine façon la qualité technique d’une implémentation, il ne garantit pas la qualité fonctionnelle. Plusieurs éléments peuvent ainsi être techniquement valides mais une fois mis ensemble ne répondent pas du tout au besoin réellement exprimé par le client. De manière un peu caricatural, le BDD va guider le développement d’une fonctionalité, tandis que le TDD guidera son implementation.

Suggestion de D. North (2003)

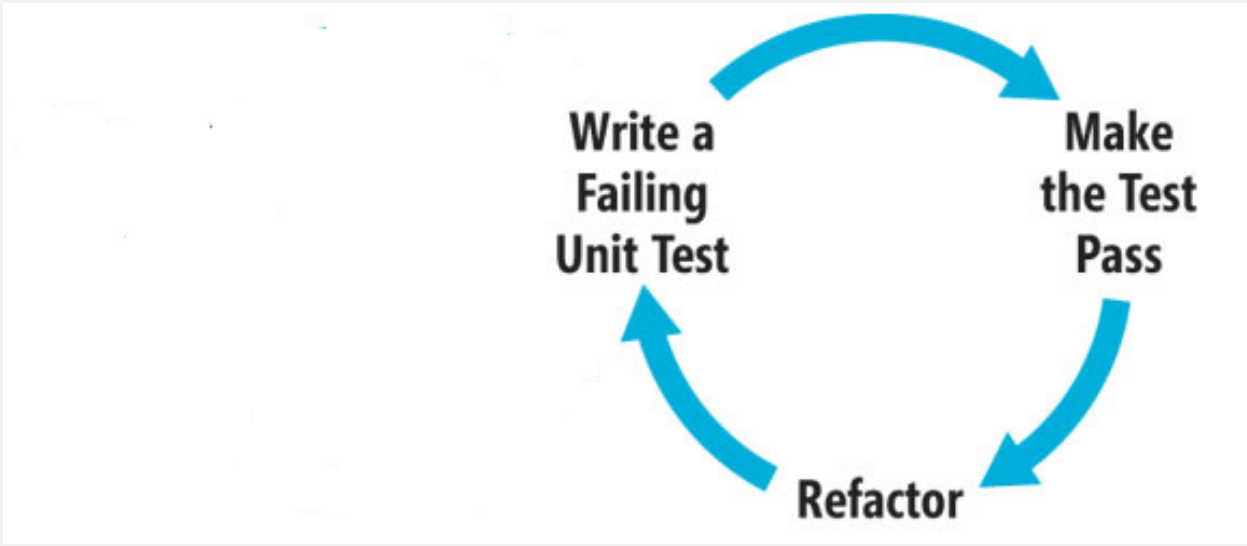
Gabarit de scénario proposé par D. North :

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


Gabarit de scénario proposé par D. North :
Scenario: name of the scenario
Given some initial context
When an event occurs
Then ensure some outcomes

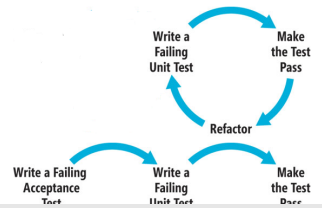
-

TDD vs. BDD : TDD et BDD



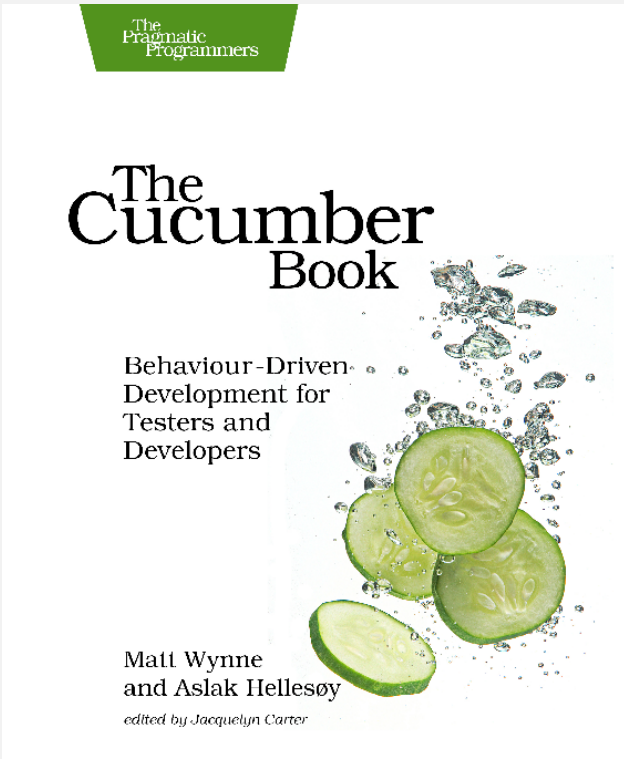
2016-11-04

Ruby =~ /[TB]DD/
└BDD = *Behavior Driven Development*
└TDD vs. BDD : TDD et BDD



-

Cucumber (Hellesoy, 2008)



Pour la spécification et l'exécution des tests systèmes !

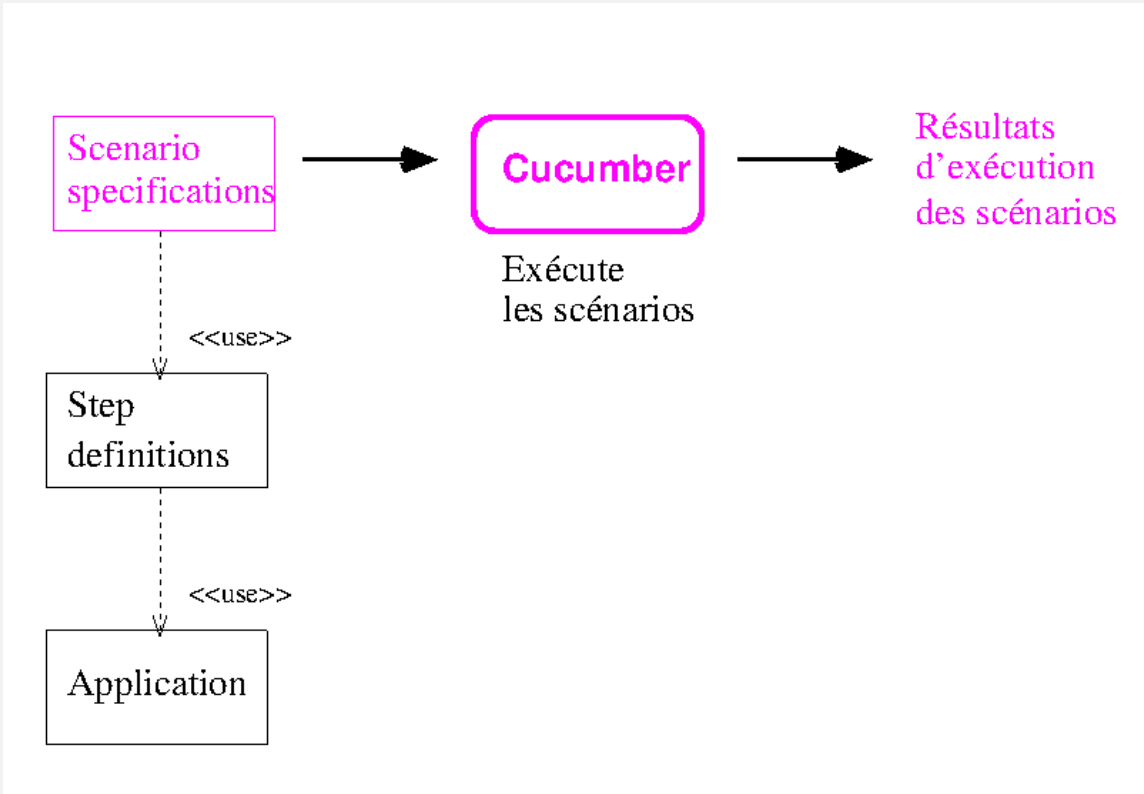
2016-11-04

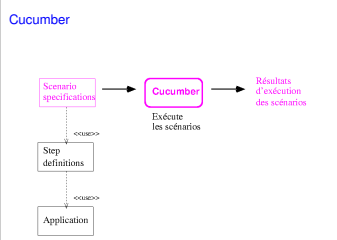
Ruby =~ /[TB]DD/
└BDD = *Behavior Driven Development*
└Cucumber (Hellesoy, 2008)

Cucumber
Pour la spécification et l'exécution des tests systèmes !

- Cette idée de scénario exécutable a alors été reprise, en Ruby, par Hellesoy, dans un outil appelé `cucumber`, que je vais vous présenter.

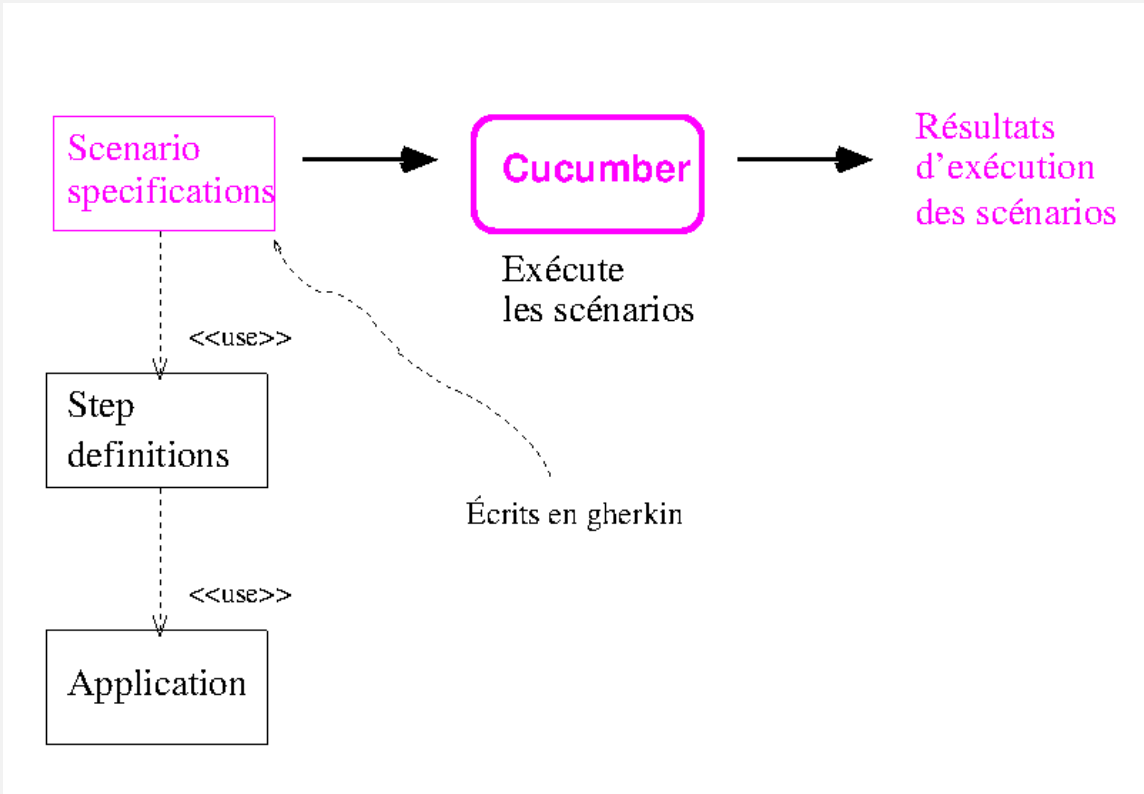
Cucumber : Principe général de fonctionnement

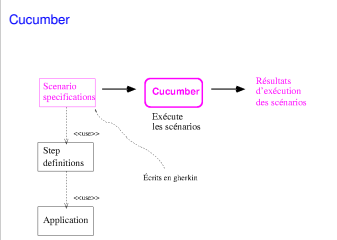




-

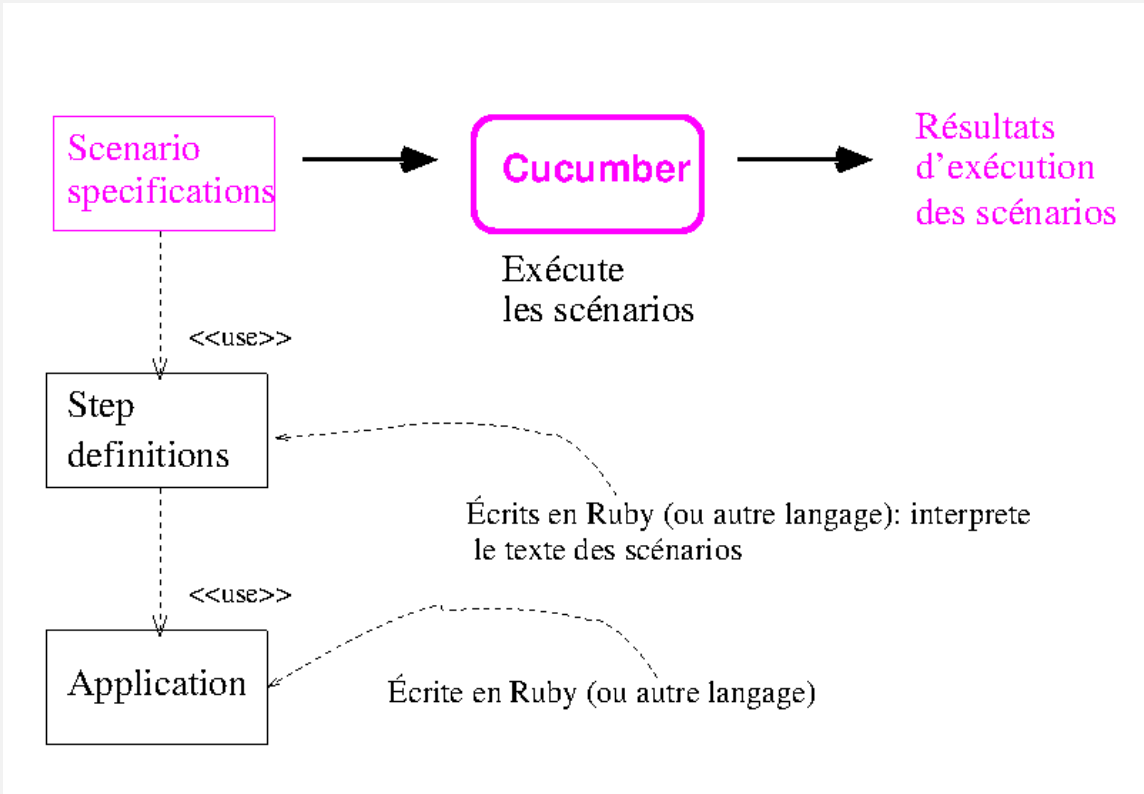
Cucumber : Principe général de fonctionnement

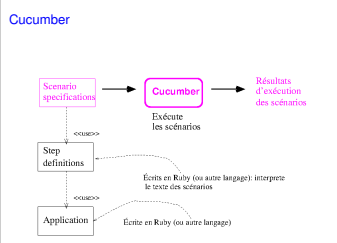




-

Cucumber : Principe général de fonctionnement





-

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

Given

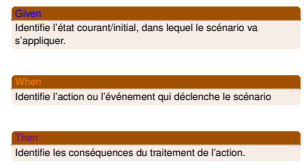
Identifie l'état courant/initial, dans lequel le scénario va s'appliquer.

When

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

Then

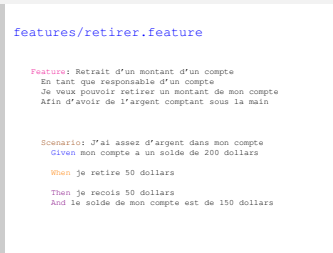
Identifie les conséquences du traitement de l'action.



-

Exemple Ruby avec `cucumber` et `gherkin` : `features/retirer.feature` (1)

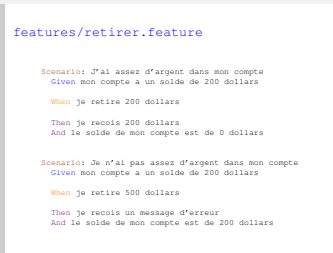
```
Feature: Retrait d'un montant d'un compte
  En tant que responsable d'un compte
  Je veux pouvoir retirer un montant de mon compte
  Afin d'avoir de l'argent comptant sous la main


  Scenario: J'ai assez d'argent dans mon compte
    Given mon compte a un solde de 200 dollars

    When je retire 50 dollars

    Then je recois 50 dollars
    And le solde de mon compte est de 150 dollars
```

- Voici un exemple pour la version simplifiée de la banque.
- `cucumber` = l'outil d'exécution des scénarios
- `gherkin` = le langage de description/spécification des scénarios
- La description de la «*feature*», au tout début, avant les scénarios, est arbitraire (texte non analysé par l'outil). Toutefois, la suggestion/convention est d'utiliser gabarit dit *Connextra format* pour la description des récits utilisateurs (*user stories*).

Exemple Ruby avec `cucumber` et `gherkin` : `features/retirer.feature` (2)

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

  When je retire 200 dollars

  Then je recois 200 dollars
  And le solde de mon compte est de 0 dollars


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

  When je retire 500 dollars

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

-

Exécution initiale… avec des étapes *pending* (suite)

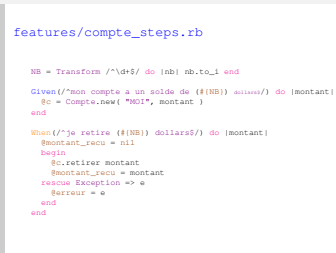
```
<<~/SeminaireTBDDD/Compte@MacBook>> $ cucumber
Feature: Retrait d'un montant d'un compte
  En tant que responsable d'un compte
  Je veux pouvoir retirer un montant de mon compte
  Afin d'avoir de l'argent comptant sous la main

  Scenario: J'ai assez d'argent dans mon compte   # features/retirer.feature:6
    Given mon compte a un solde de 200 dollars    # features/retirer.feature:7
    When je retire 50 dollars                     # features/retirer.feature:9
    Then je recois 50 dollars                     # features/retirer.feature:11
    And le solde de mon compte est de 150 dollars # features/retirer.feature:12

  Scenario: J'ai assez d'argent dans mon compte # features/retirer.feature:15
    Given mon compte a un solde de 200 dollars  # features/retirer.feature:16
    When je retire 200 dollars                  # features/retirer.feature:18
    Then je recois 200 dollars                  # features/retirer.feature:20
    And le solde de mon compte est de 0 dollars # features/retirer.feature:21

  Scenario: Je n'ai pas assez d'argent dans mon compte # features/retirer.feature:24
    Given mon compte a un solde de 200 dollars         # features/retirer.feature:25
    When je retire 500 dollars                         # features/retirer.feature:27
    Then je recois un message d'erreur                 # features/retirer.feature:29
    And le solde de mon compte est de 200 dollars      # features/retirer.feature:30

3 scenarios (3 undefined)
12 steps (12 undefined)
0m0.006s

You can implement step definitions for undefined steps with these snippets:
```

```
You can im

Given(/^mo
  pending
end

When(/^je
  pending
end

Then(/^je
  pending
end

Then(/^le
  pending
end

Then(/^je
  pending
end
```


- L'exécution initiale de ces scénarios, sans mises en oeuvre des diverses étapes, donnerait alors un résultat comme celui-ci.
- On remarque que cela dit «*3 scenarios*» et «*12 steps*». Chaque étape correspond à un *Given*, *When* ou *Then*.
- Dans l'état initial, ces étapes ne sont pas encore mises en oeuvre, et `cucumber` nous donne des suggestions quant à ce qu'il faut faire pour démarrer leur mise en oeuvre.

Mise en oeuvre (très simplifiée) : features/compte_steps.rb

```ruby
NB = Transform /^\d+$/ do |nb| nb.to_i end

Given(/^mon compte a un solde de (#{NB}) dollars$/) do |montant|
  @c = Compte.new( "MOI", montant )
end

When(/^je retire (#{NB}) dollars$/) do |montant|
  @montant_recu = nil
  begin
    @c.retirer montant
    @montant_recu = montant
  rescue Exception => e
    @erreur = e
  end
end
```

- Voici maintenant une mise en oeuvre très simplifiée des diverses étapes, qui permet d'exécuter les divers scénarios de tests avec succès.
- Dans un premier temps, voici les Given et les When, donc les pré-conditions.
- Qu'est-ce que je vais faire pour satisfaire cet antécédent. . . je vais créer un compte avec un certain solde — donc c'est comme le *setup* d'un test unitaire.

Mise en oeuvre (très simplifiée) : features/compte_steps.rb (suite)

```ruby
Then(/^le solde de mon compte est de (#{NB}) dollars$/)
      do |montant|
  expect( @c.solde ).to eq montant
end

Then(/^je recois (#{NB}) dollars$/) do |montant|
  expect( @montant_recu ).to eq montant
end

Then(/^je recois un message d'erreur$/) do
  expect( @erreur ).not_to be_nil
end
```


features/compte_steps.rb

Then(/^le solde de mon compte est de (#{NB}) dollars$/)
do |montant|
expect(@c.solde).to eq montant
end

Then(/^je recois (#{NB}) dollars$/) do |montant|
expect(@montant_recu).to eq montant
end

Then(/^je recois un message d'erreur$/) do
expect(@erreur).not_to be_nil
end

- Voici ensuite les Then, donc les post-conditions.

Mise en oeuvre (très simplifiée) : features/compte_steps.rb (suite)

```ruby
Then(/^le solde de mon compte est de (#{NB}) dollars$/)
      do |montant|
  expect/( @c.solde ).to eq montant
end

Then(/^je recois (#{NB}) dollars$/) do |montant|
  expect( @montant_recu ).to eq montant
end

Then(/^je recois un message d'erreur$/) do
  expect( @erreur ).not_to be_nil
end
```


features/compte_steps.rb

Then(/^le solde de mon compte est de (#{NB}) dollars$/)
do |montant|
expect/(@c.solde).to eq montant
end

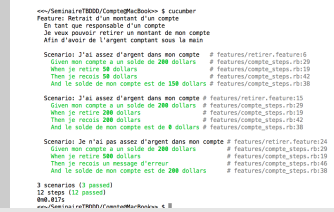
Then(/^je recois (#{NB}) dollars$/) do |montant|
expect(@montant_recu).to eq montant
end

Then(/^je recois un message d'erreur$/) do
expect(@erreur).not_to be_nil
end

- Je vous signale que les attentes sont exprimées dans le nouveau style aussi suggéré par RSpec, donc avec des `expect` plutôt qu'avec des `should`.

Exécution. . . après avoir finalisé les «étapes»

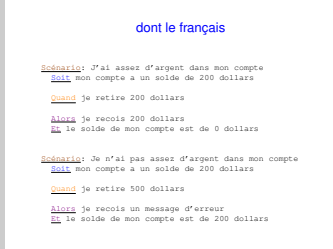
```
<<~/SeminaireTBDDD/Compte@MacBook>> $ cucumber
Feature: Retrait d'un montant d'un compte
  En tant que responsable d'un compte
  Je veux pouvoir retirer un montant de mon compte
  Afin d'avoir de l'argent comptant sous la main

  Scenario: J'ai assez d'argent dans mon compte   # features/retirer.feature:6
    Given mon compte a un solde de 200 dollars    # features/compte_steps.rb:29
    When je retire 50 dollars                     # features/compte_steps.rb:19
    Then je recois 50 dollars                     # features/compte_steps.rb:42
    And le solde de mon compte est de 150 dollars # features/compte_steps.rb:38

  Scenario: J'ai assez d'argent dans mon compte # features/retirer.feature:15
    Given mon compte a un solde de 200 dollars  # features/compte_steps.rb:29
    When je retire 200 dollars                  # features/compte_steps.rb:19
    Then je recois 200 dollars                  # features/compte_steps.rb:42
    And le solde de mon compte est de 0 dollars # features/compte_steps.rb:38

  Scenario: Je n'ai pas assez d'argent dans mon compte # features/retirer.feature:24
    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)
0m0.017s
<<~/SeminaireTBDDD/Compte@MacBook>> $
```

- Voici donc les résultats d'exécution une fois qu'on a défini ces étapes.

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 langage `gherkin` supporte une très grande quantité de langues : 37 en date de Nov. 2013 !

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           |
```


Ruby =~ /[TB]DD/
└BDD = *Behavior Driven Development*
└Le DSL de `cucumber`, `gherkin`, permet de définir des familles de cas de tests

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

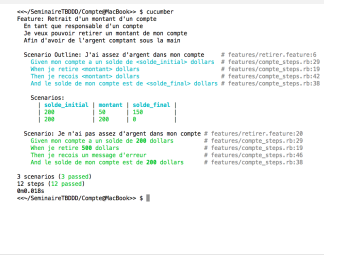
-

```
<<~/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>> $
```

2016-11-04

Ruby =~ /[TB]DD/

└─BDD = *Behavior Driven Development*



-

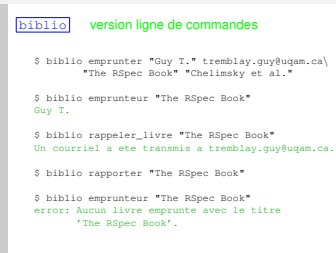
Un exemple : `biblio`

Spécification de `biblio`

-

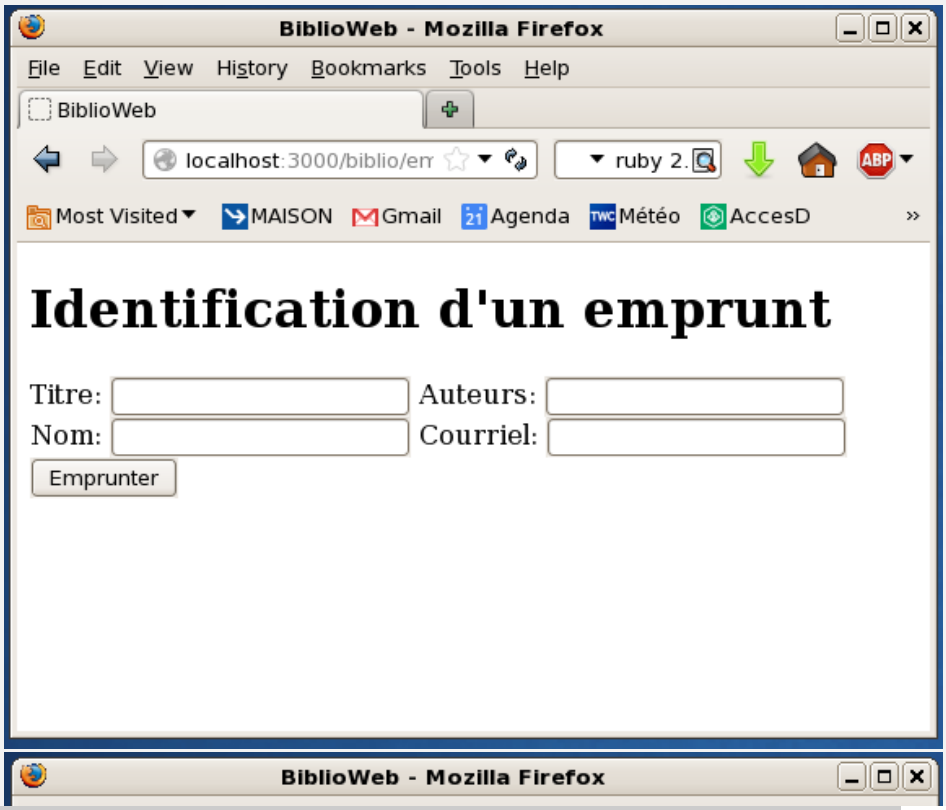
biblio — version ligne de commandes

```
$ biblio emprunter "Guy T." tremblay.guy@uqam.ca\
        "The RSpec Book" "Chelimsky et al."

$ biblio emprunteur "The RSpec Book"
Guy T.

$ biblio rappeler_livre "The RSpec Book"
Un courriel a ete transmis a tremblay.guy@uqam.ca.

$ biblio rapporter "The RSpec Book"

$ biblio emprunteur "The RSpec Book"
error: Aucun livre emprunte avec le titre
       'The RSpec Book'.
```

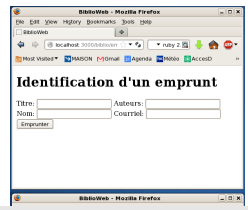



- Un exemple qui illustre ces notions, plus particulièrement dans le contexte TDD/BDD
- Pourquoi ligne de commandes ? Parce que c'est ma façon habituelle de travailler. Parce que c'est plus simple à mettre en oeuvre et à expliquer dans un exemple. Parce que c'est donc la première mise en oeuvre que j'ai développée.
- Parce que j'ai aussi déjà une version fonctionnelle en C, développée il y a quelques années par un groupe d'étudiants et qui a ensuite été utilisé comme «corpus de maintenance» dans le cours INF3135.
- Ce qui m'intéresse aujourd'hui, c'est de présenter l'allure générale de la solution, des scénarios et de certains tests, plus spécifiquement pour `rappeler_livre`.

biblio — version Web



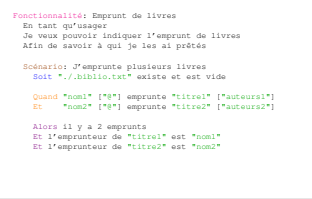
biblio version Web



- Mais, plus récemment, j'ai aussi mis en oeuvre une version, simple, avec une interface Web réalisée avec Ruby on Rails.

Spécification `cucumber` : `emprunter.feature`

```
Fonctionnalité: Emprunt de livres
  En tant qu'usager
  Je veux pouvoir indiquer l'emprunt de livres
  Afin de savoir à qui je les ai prêtés

  Scénario: J'emprunte plusieurs livres
    Soit "./.biblio.txt" existe et est vide

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

    Alors il y a 2 emprunts
    Et l'emprunteur de "titre1" est "nom1"
    Et l'emprunteur de "titre2" est "nom2"
```

-

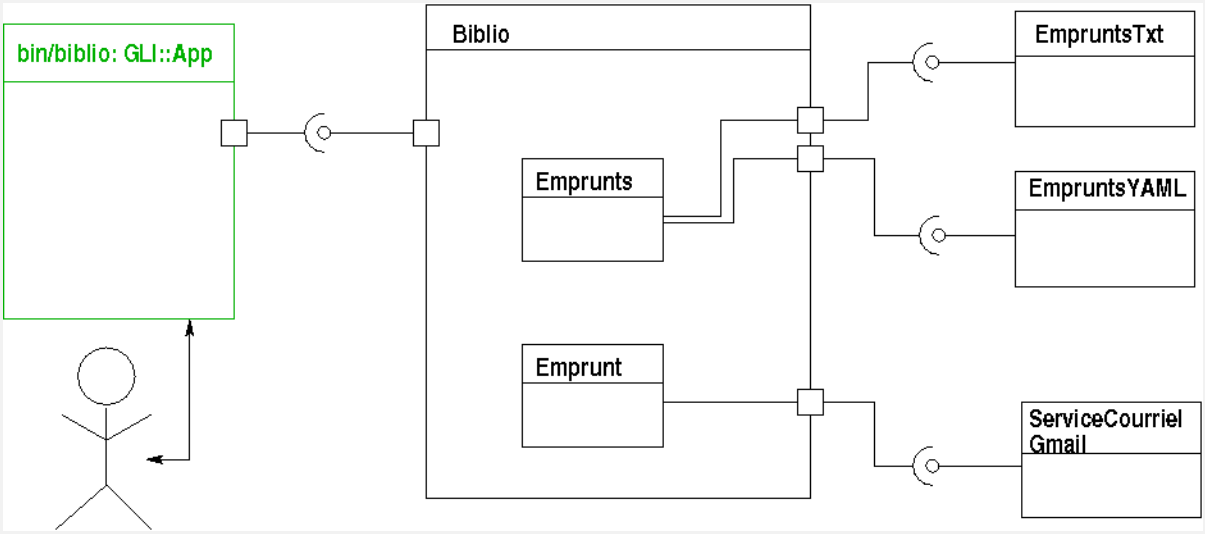
Spécification `cucumber` : `rapporter.feature`

```
Scénario: J'emprunte plusieurs livres et j'en rapporte un
  Soit "./.biblio.txt" existe et est vide

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

  Quand on rapporte "titre2"
  Et on demande l'emprunteur de "titre2"

  Alors le livre n'est pas emprunté
  Et il y a maintenant 1 emprunts
  Et l'emprunteur de "titre1" est "nom1"
```

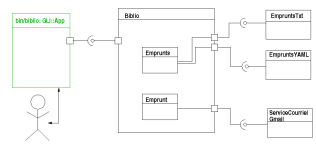

Scénario: J'emprunte plusieurs livres et j'en rapporte un
Soit "./.biblio.txt" existe et est vide
Quand "nom1" ["@"] emprunte "titre1" ["auteurs1"]
Et "nom2" ["@"] emprunte "titre2" ["auteurs2"]
Quand on rapporte "titre2"
Et on demande l'emprunteur de "titre2"
Alors le livre n'est pas emprunté
Et il y a maintenant 1 emprunts
Et l'emprunteur de "titre1" est "nom1"

-

Architecture de `biblio`

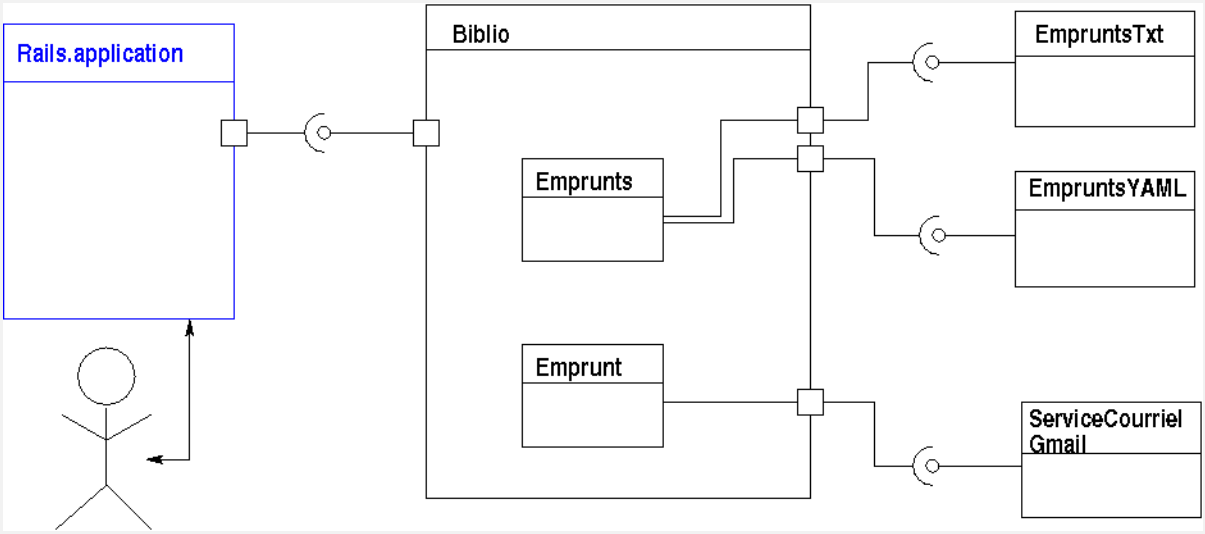


Architecture de `biblio`

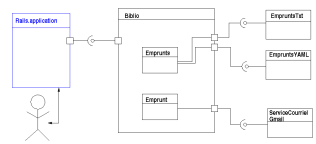


- Voici donc ce que ça donne dans le contexte de mon application `biblio` pour la gestion de prêts de livres.

Architecture de `biblio`

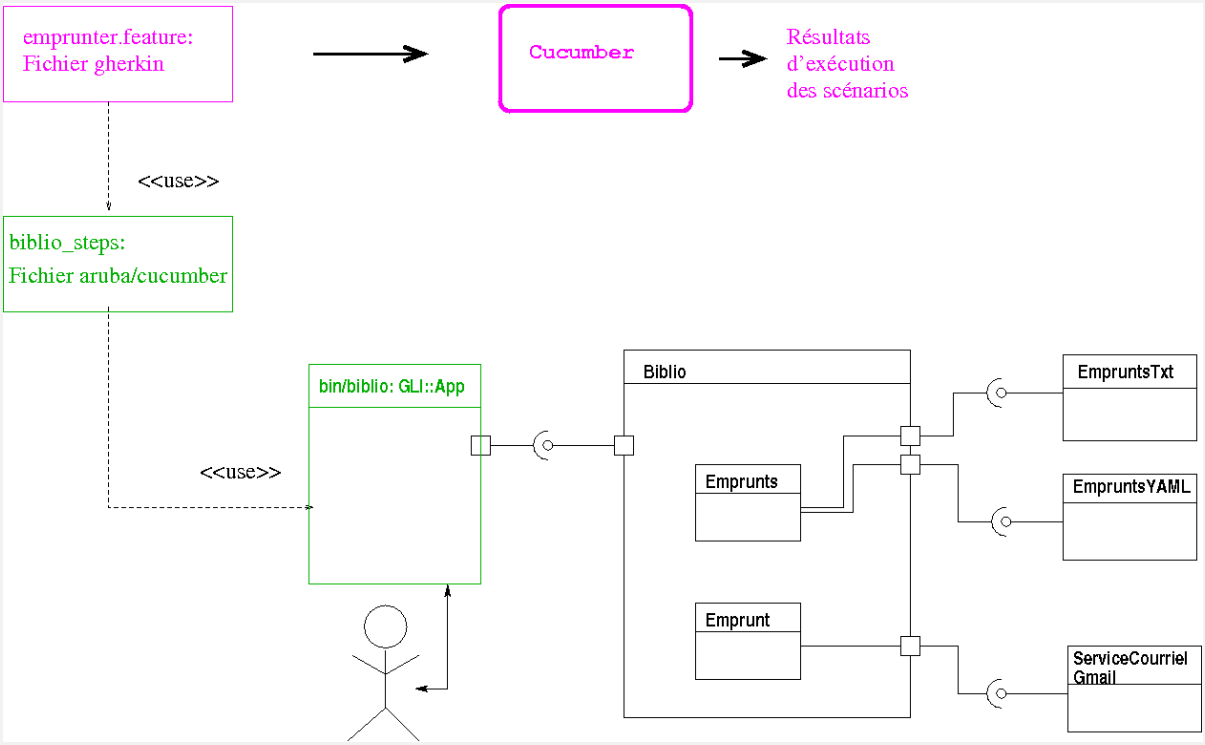


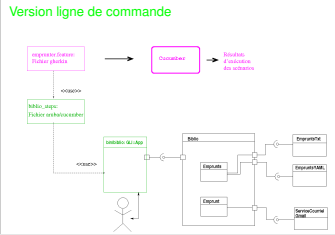
Architecture de `biblio`



- Quand on veut changer d'interface personne-machine pour une application Web, il suffit essentiellement de définir un nouveau composant, qui va utiliser les autres composants déjà définis.

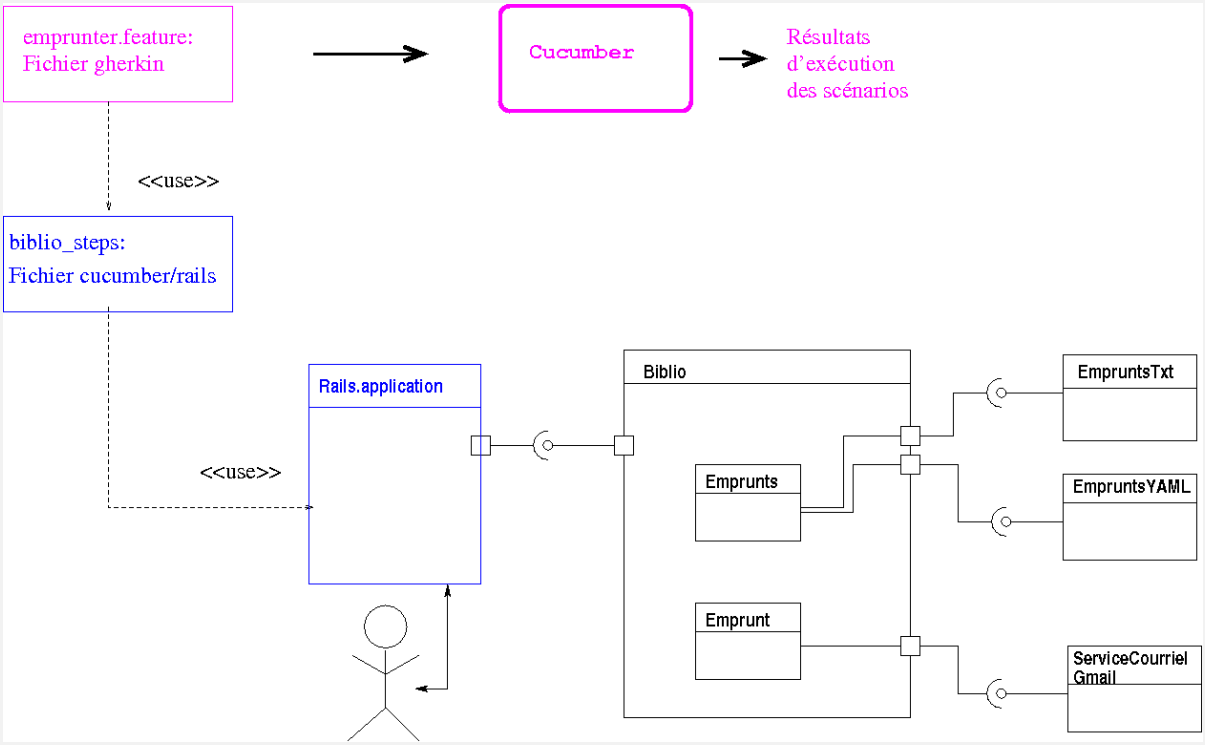
Exécution des scénarios avec Cucumber : Version ligne de commande

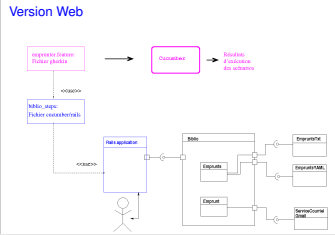




-

Exécution des scénarios avec Cucumber : Version Web

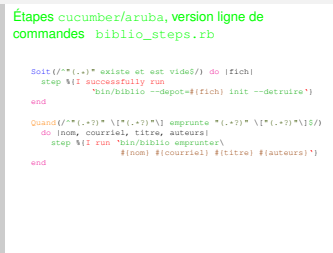




-

Étapes `cucumber/aruba`, version ligne de commandes : `biblio_steps.rb`

```
Soit(/^"(.*)" existe et est vide$/) do |fich|
  step %{I successfully run
           `bin/biblio --depot=#{fich} init --detruire`}
end

Quand(/^"(.*?)" \["(.*?)"\] emprunte "(.*?)" \["(.*?)"\]$/)
  do |nom, courriel, titre, auteurs|
    step %{I run `bin/biblio emprunter\
                  #{nom} #{courriel} #{titre} #{auteurs}`}
end
```

- Le *gem* `aruba` définit un ensemble d'étape `gherkin` prédéfinies qui permettent de définir des conditions, événements, résultats au niveau de l'exécution de commandes au niveau du *shell*.

Étapes `cucumber/rails`, version Web : `biblio_steps.rb`

```
Soit(/^"(.*)" existe et est vide$/) do |fich|
  visit "/biblio/vider"

end

Quand(/^"(.*?)" \["(.*?)"\] emprunte "(.*?)" \["(.*?)"\]$/)
  do |nom, courriel, titre, auteurs|
    visit "/biblio/emprunter"
    fill_in "Titre", :with => titre
    fill_in "Auteurs", :with => auteurs
    fill_in "Nom", :with => nom
    fill_in "Courriel", :with => courriel
    click_button "Emprunter"
end
```


Étapes `cucumber/rails`, version Web : `biblio_steps.rb`

- Mais ces étapes, si elles sont définies à un niveau d'abstractions dans les scénarios, peuvent aussi être mises en oeuvre à l'aide d'opérations sur un fureteur Web.
- Donc, la mise en oeuvre des étapes est spécifique à mon interface personne–machine.

Étapes `cucumber/aruba`, version ligne de commandes : `biblio_steps.rb`

```ruby
Alors(/^l'emprunteur de "(.*?)" est "(.*?)"$/) do |titre, nom|
  step %{I successfully run `bin/biblio emprunteur #{titre}`}
  step %{the stdout should contain "#{nom}"}
end
```


Étapes `cucumber/aruba`, version ligne de commandes : `biblio_steps.rb`

```ruby
Alors(/^l'emprunteur de "(.*?)" est "(.*?)"$/) do |titre, nom|
  step %{I successfully run `bin/biblio emprunteur #{titre}`}
  step %{the stdout should contain "#{nom}"}
end
```

-

Étapes `cucumber/rails`, version Web : `biblio_steps.rb` (suite)

```ruby
Alors(/^l'emprunteur de "(.*?)" est "(.*?)"$/) do |titre, nom|
  visit "/biblio/emprunteur"
  fill_in "Titre recherché", :with => titre
  click_button "Trouver emprunteur"
  expect(page).to have_content(nom)
end
```


Étapes cucumber/rails, version Web : biblio_steps.rb (suite)

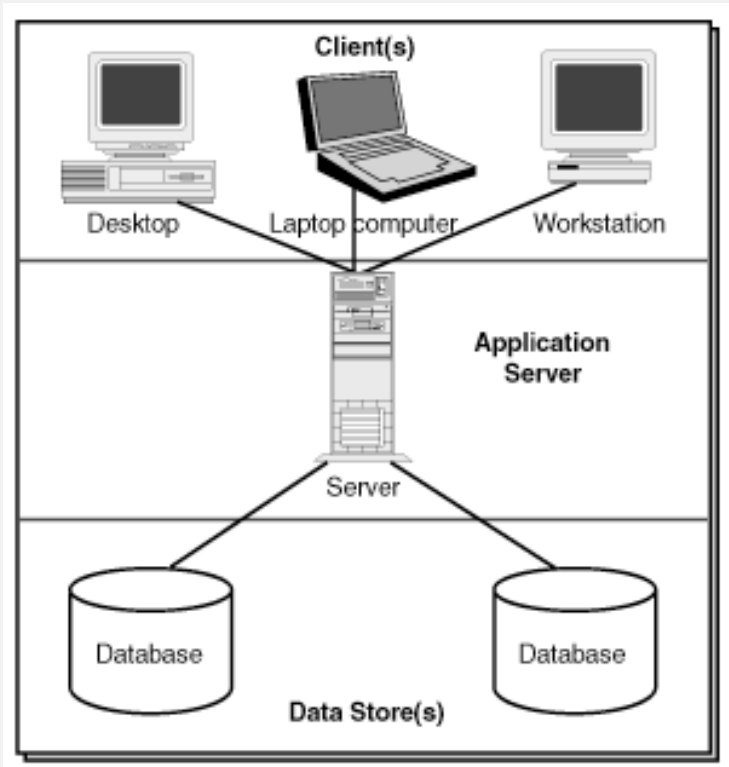
Alors(/^l'emprunteur de "(.*?)" est "(.*?)"$/) do |titre, nom|
visit "/biblio/emprunteur"
fill_in "Titre recherché", :with => titre
click_button "Trouver emprunteur"
expect(page).to have_content(nom)
end

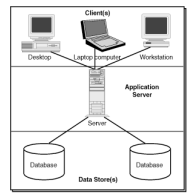
- Donc :
  - On a des scénarios qui sont abstraits
  - On a des étapes qui sont spécifiques/particulières à chacune des interfaces personne–machine.
  - On va voir que ces étapes, liées à chaque IPM, sont mises en oeuvre par des contrôleurs qui vont utiliser la même couche de modèle.

Mise en oeuvre de `biblio`

-

Architecture en couches : *three tier architecture*

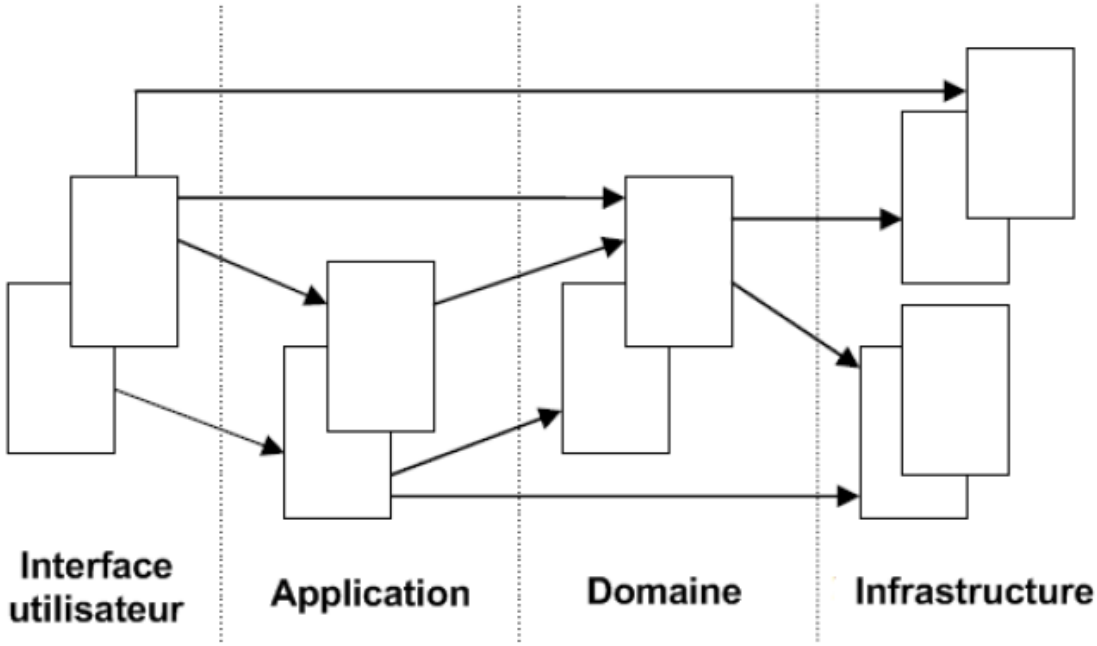


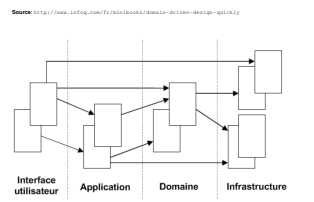


• Tout le monde connait cette architecture en couche, souvent présentée pour illustrer une bonne architecture d'un système fonctionnant avec diverses interfaces persone-machine.

Architecture en couches : *four tier architecture*

Source: `http://www.infoq.com/fr/minibooks/domain-driven-design-quickly`





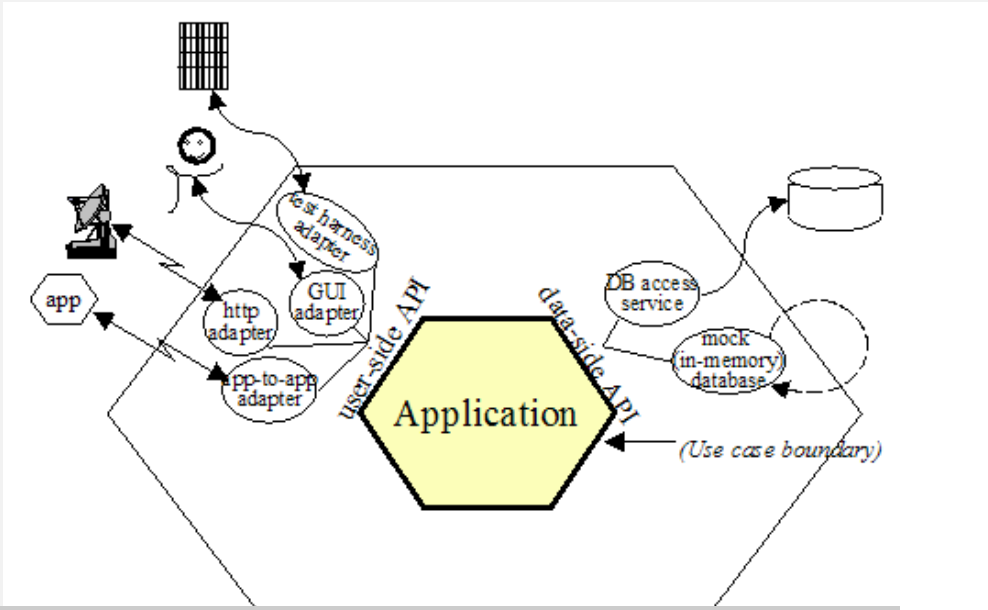
- Une autre variante d'une architecture multi-couche est celle-ci, avec 4 couches, dont une dédiée à la représentation du domaine.

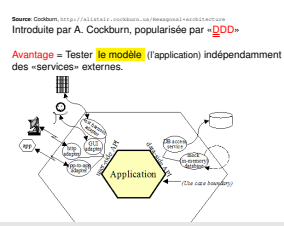
Architecture en couches : *hexagonal (ports and adapters) architecture*

Source: Cockburn, `http://alistair.cockburn.us/Hexagonal+architecture`

Introduite par A. Cockburn, popularisée par «DDD»

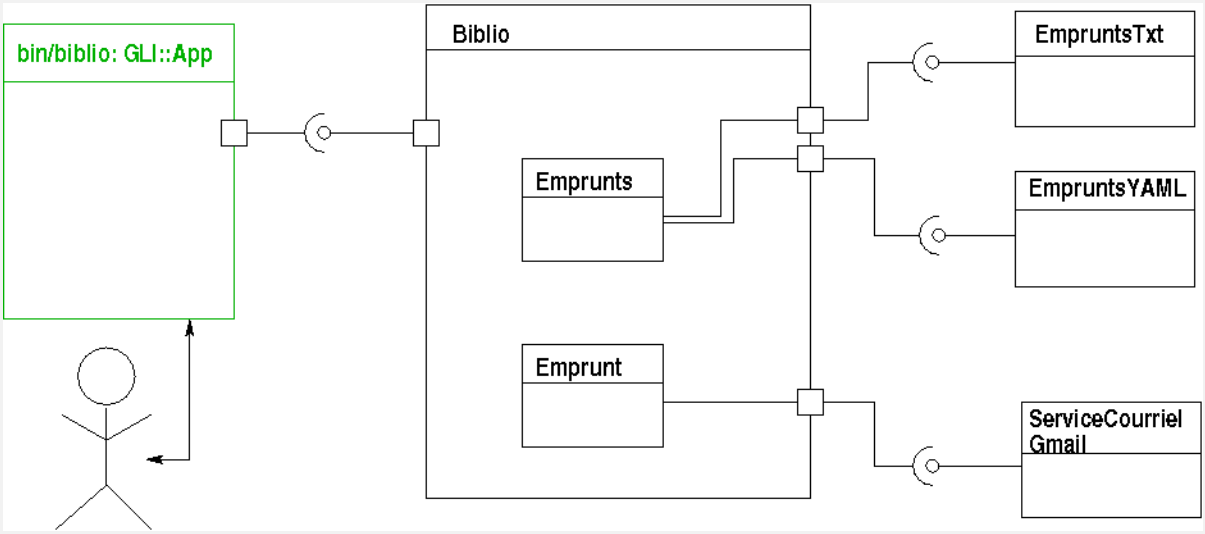
Avantage = Tester le modèle (l'application) indépendamment des «services» externes.



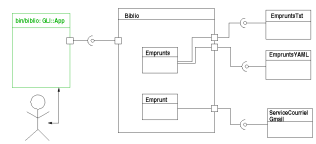


- C'est une forme d'architecture, plus générale que l'approche à quatre couches, qu'on retrouve dans les références plus récentes qui traitent de DDD, par exemple, le bouquin de Vernon, «*Implementing Domain Driven Design*».
- Patron introduit par Alistair Cockburn, circa 2004, initialement sous le nom de «*Hexagonal architecture*», puis sous le nom de «*Ports and adapters architecture*».
- Dixit Cockburn : «Create your application to work without either a UI or a database so you can run automated regression-tests against the application, work when the database becomes unavailable, and link applications together without any user involvement.»
  `http://alistair.cockburn.us/Hexagonal+architecture`
- «Advantages of this architecture : The core logic can be tested independent of outside services. It is easy to replace services by other ones that are more fit in view of changing requirements.»
  `http://www.dossier-andreas.net/software_`

Architecture de `biblio`

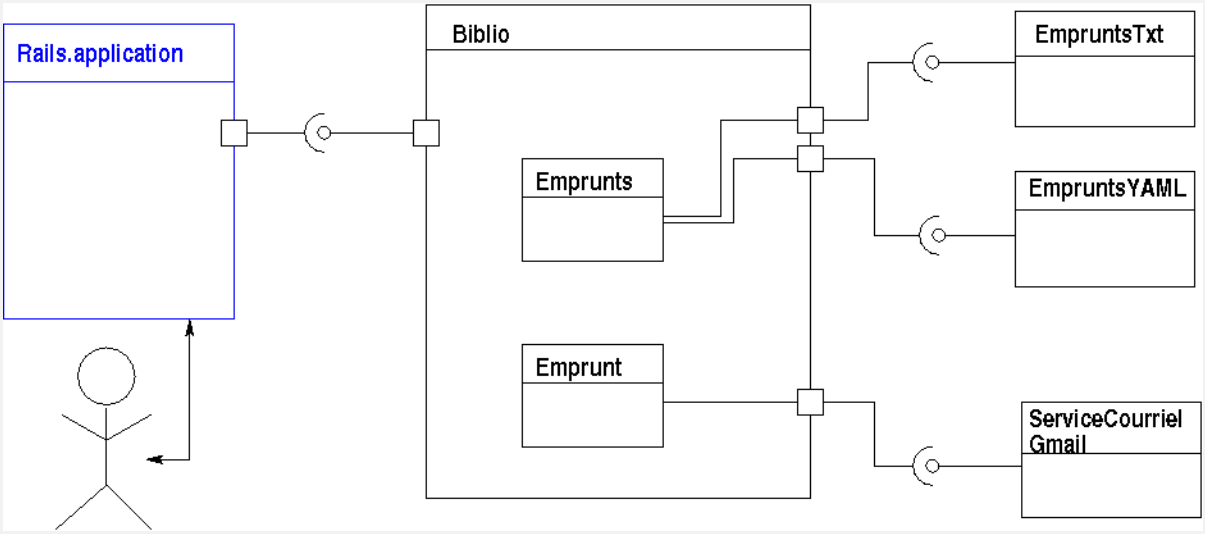


Architecture de `biblio`

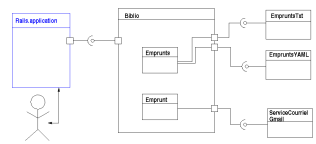


- Voici donc ce que ça donne dans le contexte de mon application `biblio` pour la gestion de prêts de livres.

Architecture de `biblio`



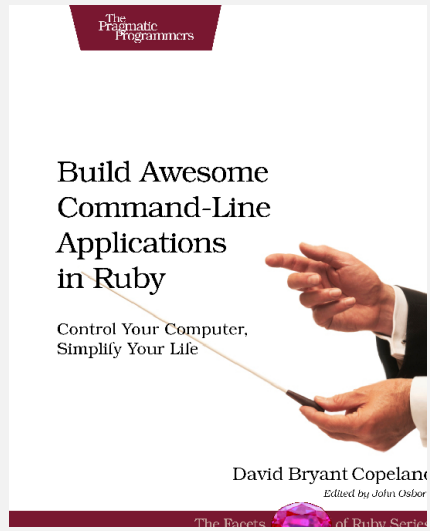
Architecture de `biblio`



- Quand on veut changer d'interface personne-machine pour une application Web, il suffit essentiellement de définir un nouveau composant, qui va utiliser les autres composants déjà définis.

Mise en oeuvre de `biblio` : version ligne de commandes

- Utilise `gli` = Gem Ruby (DSL) pour spécifier des «suites de commandes»
  - `gli` = *git like interface command line parser*



Mise en oeuvre de biblio : version ligne de commandes
Utilise gli = Gem Ruby (DSL) pour spécifier des «suites de commandes»
gli = git like interface command line parser


-

Mise en oeuvre avec `gli : bin/biblio`

```
#!/usr/bin/env ruby

...

include GLI::App

program_desc 'Programme pour la gestion de prets de livres'


# Option globale
desc 'Fichier contenant le depot'
arg_name "depot"
default_value './.biblio.txt'
flag [:depot]
```


Mise en oeuvre avec `gli : bin/biblio`

```
#!/usr/bin/env ruby

...

include GLI::App

program_desc 'Programme pour la gestion de prets de livres'


# Option globale
desc 'Fichier contenant le depot'
arg_name "depot"
default_value './.biblio.txt'
flag [:depot]
```

- Présentation descendante
- Le programme principal, la racine de l'exécutable, est `bin/biblio`
- Cet exécutable est un script Ruby, analysé et exécuté par l'interpréteur Ruby grâce au *shebang* = «# !»
- En termes d'architecture hexagonale, le fichier `bin/biblio` représente le «*primary driver*», associé aux interactions avec l'usager

Mise en oeuvre avec `gli : bin/biblio`

```
desc "Indique l'emprunt d'un livre (ou [..] stdin)"
arg_name "nom courriel titre auteurs"
command :emprunter do |c|
  c.action do |global_options,options,args|
    verifier_nb_args args, 4

    avec_biblio( global_options[:depot] ) do |bib|
      bib.emprunter( *args )
    end
  end
end
```


Mise en oeuvre avec gli : bin/biblio

desc "Indique l'emprunt d'un livre (ou [..] stdin)"
arg_name "nom courriel titre auteurs"
command :emprunter do |c|
c.action do |global_options,options,args|
verifier_nb_args args, 4

avec_biblio(global_options[:depot]) do |bib|
bib.emprunter(*args)
end
end
end

- Chaque commande de la suite est définie par un appel à `command`, suivi du nom de la commande à définir, suivi d'un bloc qui spécifie les détails de la commande — notamment, le plus important, l'action à exécuter
- Ces différentes commandes représentant, dans la terminologie MVC, les différents controleurs, qui font appel aux opérations du modèle = du domaine.

Mise en oeuvre avec `gli : bin/biblio`

```
desc "Indique le retour d'un livre"
arg_name 'titre'
command :rapporter do |c|
  c.action do |global_options,options,args|
    verifier_nb_args args, 1
    titre = args[0]

    avec_biblio( global_options[:depot] ) do |bib|
      bib.rapporter( titre )
    end
  end
end

...

exit run(ARGV)
```


Mise en oeuvre avec gli : bin/biblio

desc "Indique le retour d'un livre"
arg_name 'titre'
command :rapporter do |c|
c.action do |global_options,options,args|
verifier_nb_args args, 1
titre = args[0]

avec_biblio(global_options[:depot]) do |bib|
bib.rapporter(titre)
end
end
end

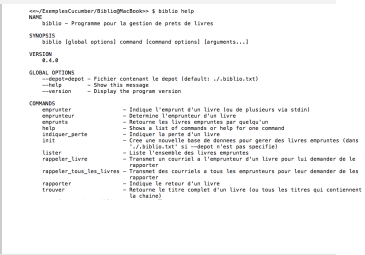
...

exit run(ARGV)

- Une fois les commandes spécifiées, il suffit ensuite simplement d'appeler la méthode `run`, définie dans `GLI::App`.

```
<<~/ExemplesCucumber/Biblio@MacBook>> $ biblio help
NAME
    biblio - Programme pour la gestion de prets de livres

SYNOPSIS
    biblio [global options] command [command options] [arguments...]

VERSION
    0.4.0

GLOBAL OPTIONS
    --depot=depot - Fichier contenant le depot (default: ./.biblio.txt)
    --help        - Show this message
    --version     - Display the program version

COMMANDS
    emprunter                - Indique l'emprunt d'un livre (ou de plusieurs via stdin)
    emprunteur               - Determine l'emprunteur d'un livre
    emprunts                 - Retourne les livres empruntes par quelqu'un
    help                     - Shows a list of commands or help for one command
    indiquer_perte           - Indiquer la perte d'un livre
    init                     - Cree une nouvelle base de donnees pour gerer des livres empruntes (dans
                               './.biblio.txt' si --depot n'est pas specifie)
    lister                   - Liste l'ensemble des livres empruntes
    rappeler_livre           - Transmet un courriel a l'emprunteur d'un livre pour lui demander de le
                               rapporter
    rappeler_tous_les_livres - Transmet des courriels a tous les emprunteurs pour leur demander de les
                               rapporter
    rapporter                - Indique le retour d'un livre
    trouver                  - Retourne le titre complet d'un livre (ou tous les titres qui contiennen
                               la chaine)
```

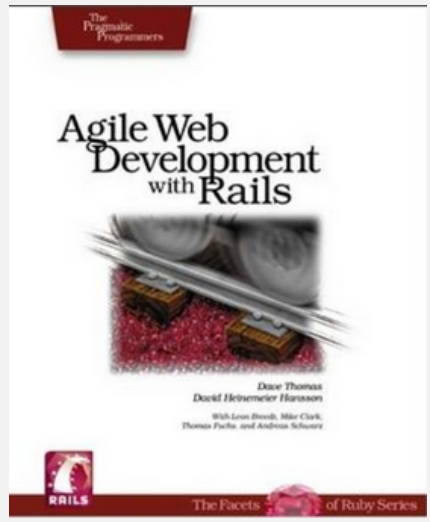

Ruby =~ /[TB]DD/
└─Un exemple : `biblio`



- Par défaut, une commande `help` est automatiquement générée, produisant la documentation illustrée dans la figure ci-haut.

Mise en oeuvre de `biblio` : version Web

- Utilise `rails` = *Framework* Ruby pour développer des applications Web



Mise en oeuvre de biblio : version Web
Utilise rails = Framework Ruby pour développer des applications Web

• Toutefois, faute de temps — ce pourrait être un séminaire complet à lui seul — je ne vous présenterai pas du tout de détails de la mise en oeuvre avec Rails.

Tests des services externes

-

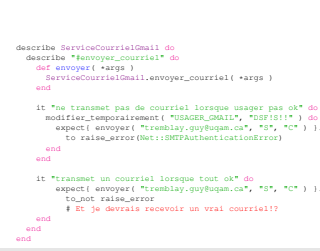
Mise en oeuvre de `ServiceCourrielGmail` : `service-courriel-gmail.rb`

```ruby
module ServiceCourrielGmail

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

end
```


- On va maintenant s'attarder aux interactions avec les entités externes, parce que c'est là que cela devient intéressant au niveau des tests.
- Voici la méthode qui permet de transmettre un courriel, en utilisant le compte `gmail` du prêteur de livres. C'est donc cette méthode qui connaît les détails, très techniques, de mise en oeuvre de l'envoi de courriel.
- Et c'est cette méthode qui doit être appelée, directement ou indirectement, par la méthode `rappeler`, qui permet d'envoyer un courriel de rappel à un emprunteur

Tests unitaires de `ServiceCourrielGmail` : `service-courriel_spec.rb`

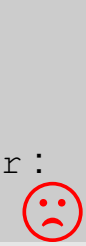
```ruby
describe ServiceCourrielGmail do
  describe "#envoyer_courriel" do
    def envoyer( *args )
      ServiceCourrielGmail.envoyer_courriel( *args )
    end

    it "ne transmet pas de courriel lorsque usager pas ok" do
      modifier_temporairement( "USAGER_GMAIL", "DSF!S!!" ) do
        expect{ envoyer( "tremblay.guy@uqam.ca", "S", "C" ) }.
          to raise_error(Net::SMTPAuthenticationError)
      end
    end

    it "transmet un courriel lorsque tout ok" do
        expect{ envoyer( "tremblay.guy@uqam.ca", "S", "C" ) }.
          to_not raise_error
          # Et je devrais recevoir un vrai courriel!?
    end
  end
end
```

- Ce qu'il est intéressant de regarder, c'est l'intérêt que cette approche a sur la forme des tests.
- Voici donc, dans un premier temps, les tests unitaires, exprimés en RSpec, pour `ServiceCourrielGmail`, donc pour le «vrai» service : on vérifie différents cas d'erreur et on vérifie «de façon non automatique» pour ce cas particulier, que l'envoi de courriel s'effectue correctement.

Mise en oeuvre de `Emprunt#rappeler` : `emprunt.rb`

```ruby
class Emprunt
  attr_reader :nom, :courriel, :titre, :auteurs
  ...

  def rappeler
    ...
    ServiceCourrielGmail.envoyer_courriel(
              courriel,
              "Retour d'un livre",
              message_courriel(titre) )
  end
  ...
end
```

class Emprunt
attr_reader :nom, :courriel, :titre, :auteurs
...
def rappeler
...
ServiceCourrielGmail.envoyer_courriel(
courriel,
"Retour d'un livre",
message_courriel(titre))
end
...
end

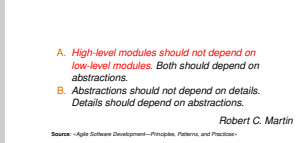
- Question : Quels sont les défauts de cette solution ?
- Un module de haut niveau — `Emprunt#rappeler` — dépend d'un module de bas niveau — `ServiceCourrielGmail.envoyer_courriel` ☹
- Ceci signifie qu'`Emprunt` est lié de façon spécifique et étroite au service de gmail, que si on décide d'utiliser un autre service, alors il faudra modifier `Emprunt#rappeler`.
- Ceci signifie aussi que pour tester `Emprunt`, et c'est cela surtout que je veux illustrer, il faut soit avoir accès à la mise en oeuvre de `ServiceCourrielGmail`, soit se définir un *stub* = méthode bidon.

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

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



- Robert C. Martin dit «*Uncle Bob*».

Mise en oeuvre de `Emprunt#rappeler` : `emprunt.rb`

```
class Emprunt
  attr_reader :nom, :courriel, :titre, :auteurs
  ...

  def rappeler
    ...
    ServiceCourrielGmail.envoyer_courriel(
              courriel,
              "Retour d'un livre",
              message_courriel(titre) )
  end
  ...
end
```

⇒ Ne respecte pas le DIP 

class Emprunt
  attr_reader :nom, :courriel, :titre, :auteurs
  ...

  def rappeler
    ...
    ServiceCourrielGmail.envoyer_courriel(
              courriel,
              "Retour d'un livre",
              message_courriel(titre))
  end
  ...
end

⇒ Ne respecte pas le DIP

-

Mise en oeuvre de `Emprunt#rappeler` : `emprunt.rb` (bis)

Dans `bin/biblio` :

```
Biblio::ServicesExternes.courriel = ServiceCourrielGmail
```

Dans `biblio/emprunt.rb` :

```
class Emprunt
  attr_reader :nom, :courriel, :titre, :auteurs
  ...

  def rappeler
    ...
    ServicesExternes.courriel.envoyer_courriel(
              courriel,
              "Retour d'un livre",
              message_courriel(titre) )
  end
  ...
end
```


- Dans le programme principal (`bin/biblio`), on définit une variable globale qui identifie quel service d'envoi de courriel doit être utilisé.
- Dans la méthode `rappeler`, on réfère à cette variable pour obtenir le nom du service à utiliser, objet/module sur lequel appelle alors la méthode `envoyer_courriel`.

Mise en oeuvre de `Emprunt#rappeler` : `emprunt.rb` (bis)

Dans `bin/biblio` :

```
Biblio::ServicesExternes.courriel = ServiceCourrielGmail
```

Dans `biblio/emprunt.rb` :

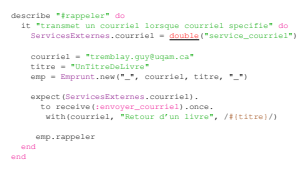
```
class Emprunt
  attr_reader :nom, :courriel, :titre, :auteurs
  ...

  def rappeler
    ...
    ServicesExternes.courriel.envoyer_courriel(
              courriel,
              "Retour d'un livre",
              message_courriel(titre) )
  end
  ...
end
```

⇒ Respecte le DIP 

- C'est une forme «d'injection de dépendances» : style *setter injection*.
- Cela peut aussi être vu comme une forme de *service locator* = «*This pattern uses a central registry known as the "service locator", which on request returns the information necessary to perform a certain task*» : `http://en.wikipedia.org/wiki/Service_locator_pattern`
- Autre nom : La variable globale joue le rôle d'un registre des services (*service registry*).

Tests unitaires de `Emprunt#rappeler` : `emprunt_spec.rb`

```ruby
describe "#rappeler" do
  it "transmet un courriel lorsque courriel specifie" do
    ServicesExternes.courriel = double("service_courriel")

    courriel = "tremblay.guy@uqam.ca"
    titre = "UnTitreDeLivre"
    emp = Emprunt.new("_", courriel, titre, "_")

    expect(ServicesExternes.courriel).
      to receive(:envoyer_courriel).once.
       with(courriel, "Retour d'un livre", /#{titre}/)

     emp.rappeler
  end
end
```

- Voici maintenant les tests pour la méthode `rappeler` de la classe `Emprunt`, qui utilise le service d'envoi de courriel.

On n'a pas besoin de tester à nouveau l'envoi effectif de courriel, car cela a déjà été fait dans les tests pour `ServiceCourrielGmail`.

Et ici, dans ce test, on ne veut pas non plus dépendre spécifiquement du service de gmail, car rien nous dit que c'est ce service qui est utilisé, i.e., un tout autre service pourrait très bien être utilisé à la place. Ce n'est pas à `rappeler` à savoir cela.

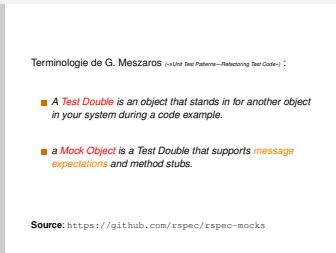
Ici, on a simplement besoin de s'assurer que le service, quel qu'il soit, reçoive la demande appropriée. C'est ce qu'on fait à l'aide d'un «*test double*» et à l'aide d'attentes explicites (*expectations*) sur le comportement observé.

Test doubles & mocks

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

- *A Test Double is an object that stands in for another object in your system during a code example.*
- *a Mock Object is a Test Double that supports message expectations and method stubs.*

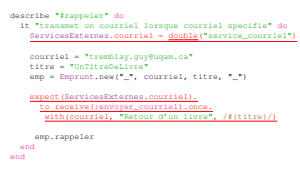
Source: `https://github.com/rspec/rspec-mocks`



•

Tests unitaires de `Emprunt#rappeler` : `emprunt_spec.rb`

```ruby
describe "#rappeler" do
  it "transmet un courriel lorsque courriel specifie" do
    ServicesExternes.courriel = double("service_courriel")

    courriel = "tremblay.guy@uqam.ca"
    titre = "UnTitreDeLivre"
    emp = Emprunt.new("_", courriel, titre, "_")

    expect(ServicesExternes.courriel).
      to receive(:envoyer_courriel).once.
       with(courriel, "Retour d'un livre", /#{titre}/)

     emp.rappeler
  end
end
```

- Lorsque le test se termine, après avoir appelé `rappeler`, il faut qu'un appel à la méthode `envoyer_courriels` ait été reçu par le `ServicesExternes.courriel` avec les arguments indiqués, sinon le test échouera.

```ruby
  def rappeler
    ...
    ServicesExternes.courriel::envoyer_courriel(
              courriel,
              "Retour d'un livre",
              message_courriel(titre) )
  end
```

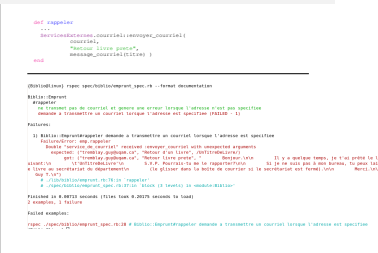
```
{Biblio@linux} rspec spec/biblio/emprunt_spec.rb --format documentation

Biblio::Emprunt
  #rappeler
    ne transmet pas de courriel et genere une erreur lorsque l'adresse n'est pas specifiee
    demande a transmettre un courriel lorsque l'adresse est specifiee

Finished in 0.00725 seconds (files took 0.22055 seconds to load)
2 examples, 0 failures
```


- Ici, on voit ce qui est affiché/indiqué si l'*expectation* est satisfaite — en format `documentation`.

```ruby
  def rappeler
    ...
    ServicesExternes.courriel::envoyer_courriel(
             courriel,
             "Retour livre prete",
             message_courriel(titre) )
  end
```

```
{Biblio@linux} rspec spec/biblio/emprunt_spec.rb --format documentation

Biblio::Emprunt
  #rappeler
    ne transmet pas de courriel et genere une erreur lorsque l'adresse n'est pas specifiee
    demande a transmettre un courriel lorsque l'adresse est specifiee (FAILED - 1)

Failures:

  1) Biblio::Emprunt#rappeler demande a transmettre un courriel lorsque l'adresse est specifiee
     Failure/Error: emp.rappeler
       Double "service_de_courriel" received :envoyer_courriel with unexpected arguments
         expected: ("tremblay.guy@uqam.ca", "Retour d'un livre", /UnTitreDeLivre/)
              got: ("tremblay.guy@uqam.ca", "Retour livre prete", "        Bonjour.\n\n        Il y a quelque temps, je t'ai pr
uivant:\n        \t'UnTitreDeLivre'\n        S.V.P. Pourrais-tu me le rapporter?\n\n        Si je ne suis pas à mon bureau, tu
e livre au secrétariat du département\n        (le glisser dans la boîte de courrier si le secrétariat est fermé).\n\n
   Guy T.\n")
       # ./lib/biblio/emprunt.rb:76:in `rappeler'
       # ./spec/biblio/emprunt_spec.rb:37:in `block (3 levels) in <module:Biblio>'

Finished in 0.00713 seconds (files took 0.20175 seconds to load)
2 examples, 1 failure

Failed examples:

rspec ./spec/biblio/emprunt_spec.rb:28 # Biblio::Emprunt#rappeler demande a transmettre un courriel lorsque l'adresse est speci
```


Ruby =~ /[TB]DD/
└─Un exemple : `biblio`



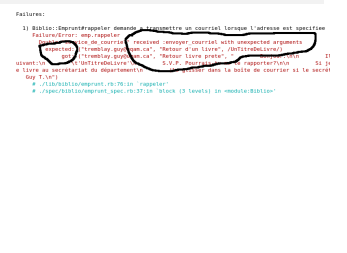
- Ici, on voit ce qui est affiché/indiqué si l'*expectation* n'est pas satisfaite — par exemple, parce que le sujet du message indiqué dans la mise en oeuvre de `Emprunt#rappeler` n'est pas le bon sujet : `"Retour livre prete"` au lieu de `"Retour d'un livre"`.

```
Failures:

  1) Biblio::Emprunt#rappeler demande a transmettre un courriel lorsque l'adresse est specifiee
     Failure/Error: emp.rappeler
       Double "service_de_courriel" received :envoyer_courriel with unexpected arguments
         expected: ("tremblay.guy@uqam.ca", "Retour d'un livre", /UnTitreDeLivre/)
              got: ("tremblay.guy@uqam.ca", "Retour livre prete", "                Bonjour!\n\n        Il
uivant:\n                \t'UnTitreDeLivre'\n                S.V.P. Pourrais-tu me le rapporter?\n\n        Si je
e livre au secrétariat du département\n                (le glisser dans la boite de courrier si le secrét
     Guy T.\n")
       # ./lib/biblio/emprunt.rb:76:in `rappeler'
       # ./spec/biblio/emprunt_spec.rb:37:in `block (3 levels) in <module:Biblio>'
```

2016-11-04

Ruby =~ /[TB]DD/

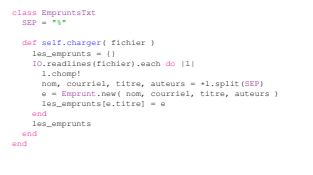
└─Un exemple : `biblio`



-

Mise en oeuvre de `emprunts-txt.rb`

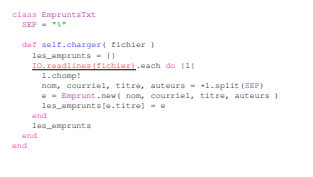
```ruby
class EmpruntsTxt
  SEP = "%"

  def self.charger( fichier )
    les_emprunts = {}
    IO.readlines(fichier).each do |l|
      l.chomp!
      nom, courriel, titre, auteurs = *l.split(SEP)
      e = Emprunt.new( nom, courriel, titre, auteurs )
      les_emprunts[e.titre] = e
    end
    les_emprunts
  end
end
```

- Voici un autre exemple, cette fois pour le service externe qui charge en mémoire le contenu de la base de données, lorsque celle-ci est en format textuelle.

Mise en oeuvre de `emprunts-txt.rb`

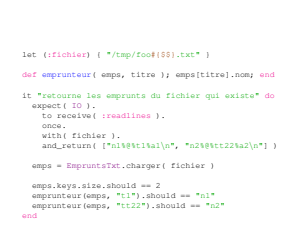
```ruby
class EmpruntsTxt
  SEP = "%"

  def self.charger( fichier )
    les_emprunts = {}
    IO.readlines(fichier).each do |l|
      l.chomp!
      nom, courriel, titre, auteurs = *l.split(SEP)
      e = Emprunt.new( nom, courriel, titre, auteurs )
      les_emprunts[e.titre] = e
    end
    les_emprunts
  end
end
```

- Les détails ne sont pas importants. L'aspect important sur lequel j'insiste est que la méthode `IO.readlines` est utilisée pour lire le contenu du fichier contenant cette base de données, méthode qui doit donc faire un accès externe à un fichier.
- Or, en autant que c'est possible, notamment pour des raisons de performance, il est préférable de limiter les accès à des fichiers externes dans les tests.

Tests unitaires de `emprunts_pour` : `emprunts-txt_spec.rb`

```ruby
let (:fichier) { "/tmp/foo#{$$}.txt" }

def emprunteur( emps, titre ); emps[titre].nom; end

it "retourne les emprunts du fichier qui existe" do
  expect( IO ).
    to receive( :readlines ).
    once.
    with( fichier ).
    and_return( ["n1%@%t1%a1\n", "n2%@%tt22%a2\n"] )

  emps = EmpruntsTxt.charger( fichier )

  emps.keys.size.should == 2
  emprunteur(emps, "t1").should == "n1"
  emprunteur(emps, "tt22").should == "n2"
end
```

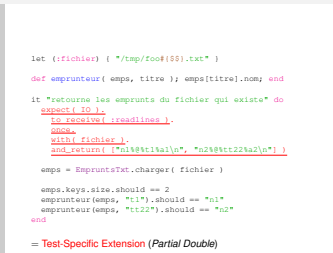



- Voici donc une façon de définir un test pour `charger` qui fait en sorte de ne pas avoir d'accès à un fichier externe, et ce en utilisant un «*partial double* », ce qu'on appelle aussi «*a tests-specific extension*» = «*an extension of a real object in a system that is instrumented with test-double like behaviour in the context of a test*» : `https://github.com/rspec/rspec-mocks`.
- Donc, on dit à l'objet `IO` que temporairement, le temps du test, il doit modifier son comportement pour la méthode `readlines` de façon à ce qu'elle retourne le tableau indiqué si elle reçoit les argument indiqués. Sinon, une erreur doit être signalée si cette méthode n'est pas appelée avec ces arguments.

Tests unitaires de `emprunts_pour` : `emprunts-txt_spec.rb`

```ruby
let (:fichier) { "/tmp/foo#{$$}.txt" }

def emprunteur( emps, titre ); emps[titre].nom; end

it "retourne les emprunts du fichier qui existe" do
  expect( IO ).
    to receive( :readlines ).
    once.
    with( fichier ).
    and_return( ["n1%@%t1%a1\n", "n2%@%tt22%a2\n"] )

  emps = EmpruntsTxt.charger( fichier )

  emps.keys.size.should == 2
  emprunteur(emps, "t1").should == "n1"
  emprunteur(emps, "tt22").should == "n2"
end
```

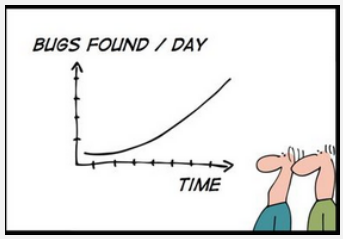
= Test-Specific Extension (*Partial Double*)

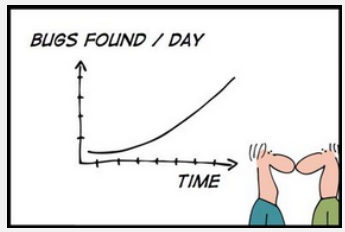


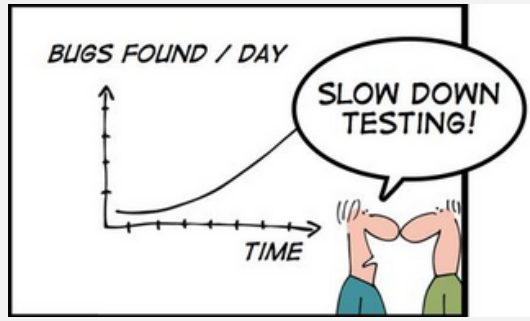
- Une telle extension, temporaire et spécifique au test, de `IO.readlines` est beaucoup plus simple à définir… que si on avait dû créer un fichier externe `foo.txt` et y mettre comme contenu les lignes désirées.
- En fait, noobstant l'aspect performance, cela permet aussi une meilleure localité des informations pour ce test — pour que le contenu soit clairement local au test, il aurait fallu créer ce fichier au moment du test (ouverture en création/écriture), à partir du contenu désiré, et ensuite lire son contenu (ouverture, implicite, en lecture avec `readlines`).

Conclusion

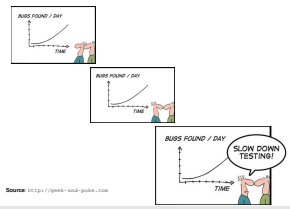
Les tests développés tôt sont cruciaux pour trouver rapidement les bogues







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



-

Les tests développés tôt sont cruciaux pour trouver rapidement les bogues

«*I find that weeks of coding and testing can save me hours of planning*».

James R. Coplien («Why Most Unit Testing is Waste»)

«*I find that weeks of coding and testing can save me hours of planning*».

James R. Coplien («Why Most Unit Testing is Waste»)

-

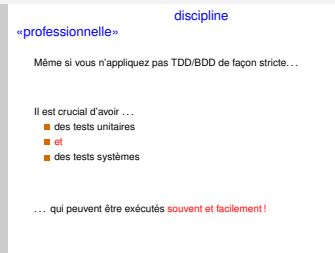
Il est important d'avoir une discipline de travail qui est «professionnelle»

Même si vous n'appliquez pas TDD/BDD de façon stricte. . .

Il est crucial d'avoir . . .

- des tests unitaires
- et
- des tests systèmes

. . . qui peuvent être exécutés souvent et facilement !



• Plus aucune raison de nos jours d'avoir du code sans tests unitaires s'exécutant de façon automatique. . . ni du code sans tests systèmes.
• Il existe aussi des tests pour automatiser les applications Web, des *drivers* de fureteur — montrer exemple si connecté ?
• Apprenez de nouveaux langages. . . et de nouveaux outils, dont les outils de test

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`

[54m00s à 54m38]
[59m00 à 1h00m40]

discipline
«professionnelle»
Extrait vidéo d'«*Uncle Bob*» (Robert C. Martin) :
https://www.youtube.com/watch?v=YX3iRjKj7C0
[54m00s à 54m38]
[59m00 à 1h00m40]

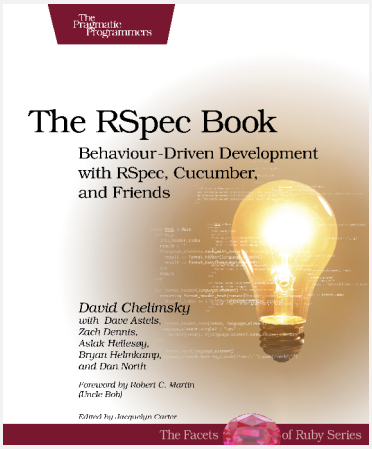
-

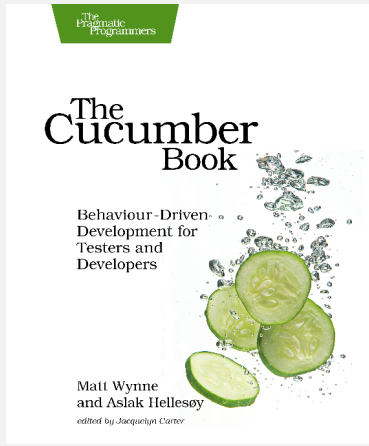
Pour en savoir plus

Pour en savoir plus

-

Pour en savoir plus





2016-11-04

Ruby =~ /[TB]DD/
└Conclusion
└Pour en savoir plus

-

Pour en savoir plus

K. Beck.
Test-Driven Development—By Example.
Addison-Wesley, Reading, MA, 2003.

D. Chelimsky, D. Astels, Z. Dennis, A. Hellesoy, B. Helmkamp, and D. North.
The RSpec Book : Behaviour Driven Development with RSpec, Cucumber, and Friends.
Pragmatic Bookshelf, 2010.

D.B. Copeland.
Build Awesome Command-Line Applications in Ruby : Control Your Computer, Simplify Your Life.
Pragmatic Bookshelf, 2012.

E. Evans.
Domain-Driven Design—Tackling Complexity in the Heart of Software.
Addison-Wesley, 2004.

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

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

K. Beck.
Test-Driven Development—By Example.
Addison-Wesley, Reading, MA, 2003.

D. Chelimsky, D. Astels, Z. Dennis, A. Hellesoy, B. Helmkamp, and D. North.
The RSpec Book : Behaviour Driven Development with RSpec, Cucumber, and Friends.
Pragmatic Bookshelf, 2010.

D.B. Copeland.
Build Awesome Command-Line Applications in Ruby : Control Your Computer, Simplify Your Life.
Pragmatic Bookshelf, 2012.

E. Evans.
Domain-Driven Design—Tackling Complexity in the Heart of Software.
Addison-Wesley, 2004.

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

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

Questions ?

Questions ?

-