声明：

1.本课程所用PPT和作业内容源于edx：BerkeleyX： Engineering Software as a Service。

2. 本课程所用PPT和作业内容经过了部分删除和修改。

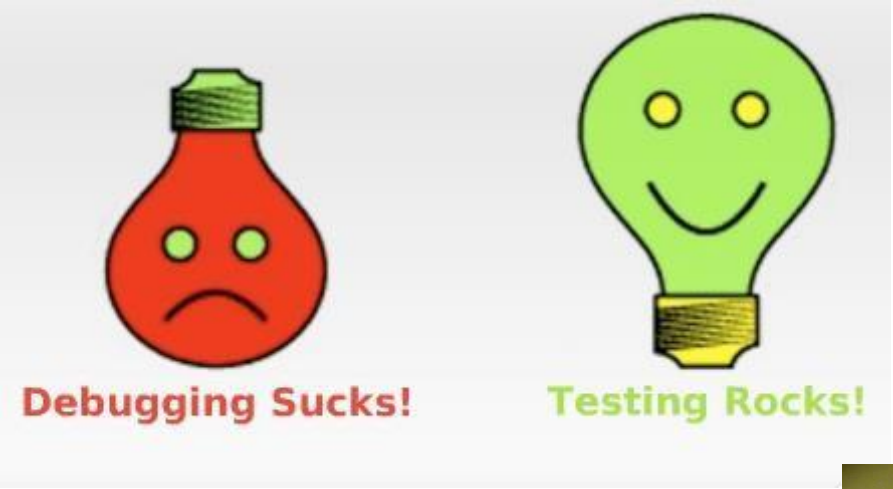
3.本课程所用代码管理平台和实验部署平台涉及部分互联网开源内容。

软件测试：测试驱动的开发

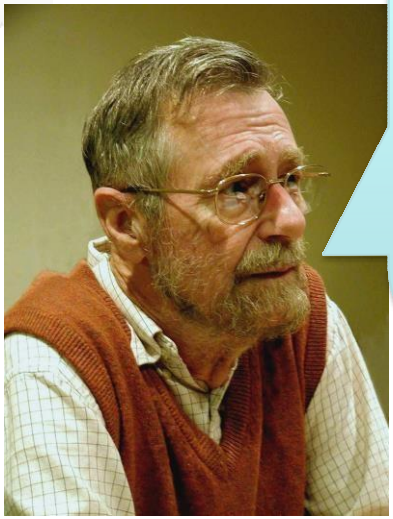
(Engineering Software as a Service §8)

RSpec和单元测试简介

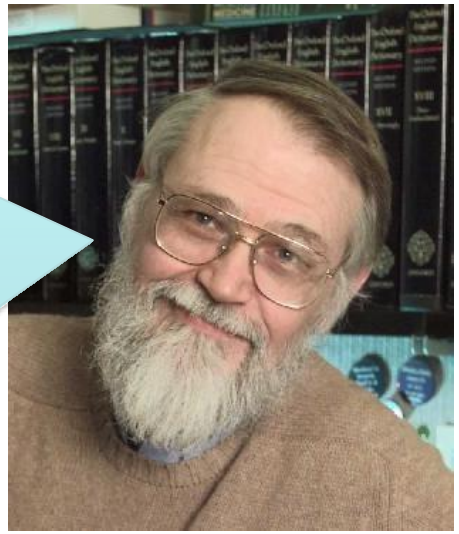
(Engineering Software as a Service §8.1)



测试永远不能证明软件中没有错误，而只能证实它们的存在



调试的难度是编写代码的两倍。因此，如果你尽可能巧妙地编写代码，那么根据定义，你还是不足够聪明能调试这些代码。



今天的测试

- 以前
  - 开发人员完成代码，进行一些特殊的测试
  - “任务扔给质量保障部门 [QA]”
    - QA部门员工手工操作软件
- 今天/敏捷
  - 测试是每个敏捷迭代的一部分
  - 开发人员测试他们自己的代码
  - 测试工具和过程高度自动化
    - QA/测试小组改进可测试性和工具

今天的测试

- 以前
  - 开发人员完成代码，进行一些特殊的测试
  - 测试是一个单独建造的一部分
  - 开发人员测试他们自己的代码
  - 测试工具和过程高度自动化
    - QA/测试小组改进可测试性和工具

软件质量是一个好过程保障的结果，
而不是一个特定小组的责任

为什么不测试的迷思

- 为什么不测试之迷思 #1:

  测试时间太长了: “*AKA 我很差*"
- 为什么不测试之迷思 #2:

  这部分代码是不可测试的: “AKA 我代码很烂"
- 为什么不测试之迷思 #3:

  我不允许这么做: “AKA 我的公司很烂"
- 为什么不测试之迷思 #4:

  这段代码不会存在很长时间: "AKA 我对生活的态度很糟糕

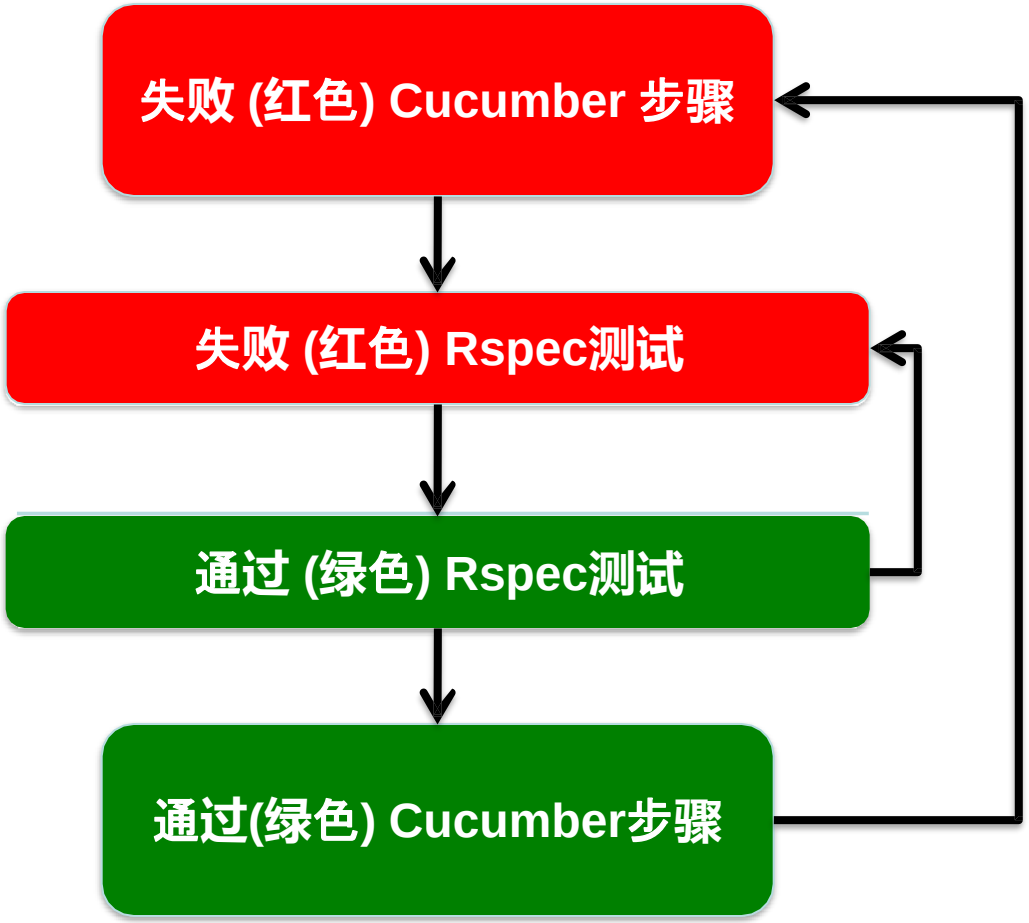
AKA- Also Known As

BDD+TDD：概况

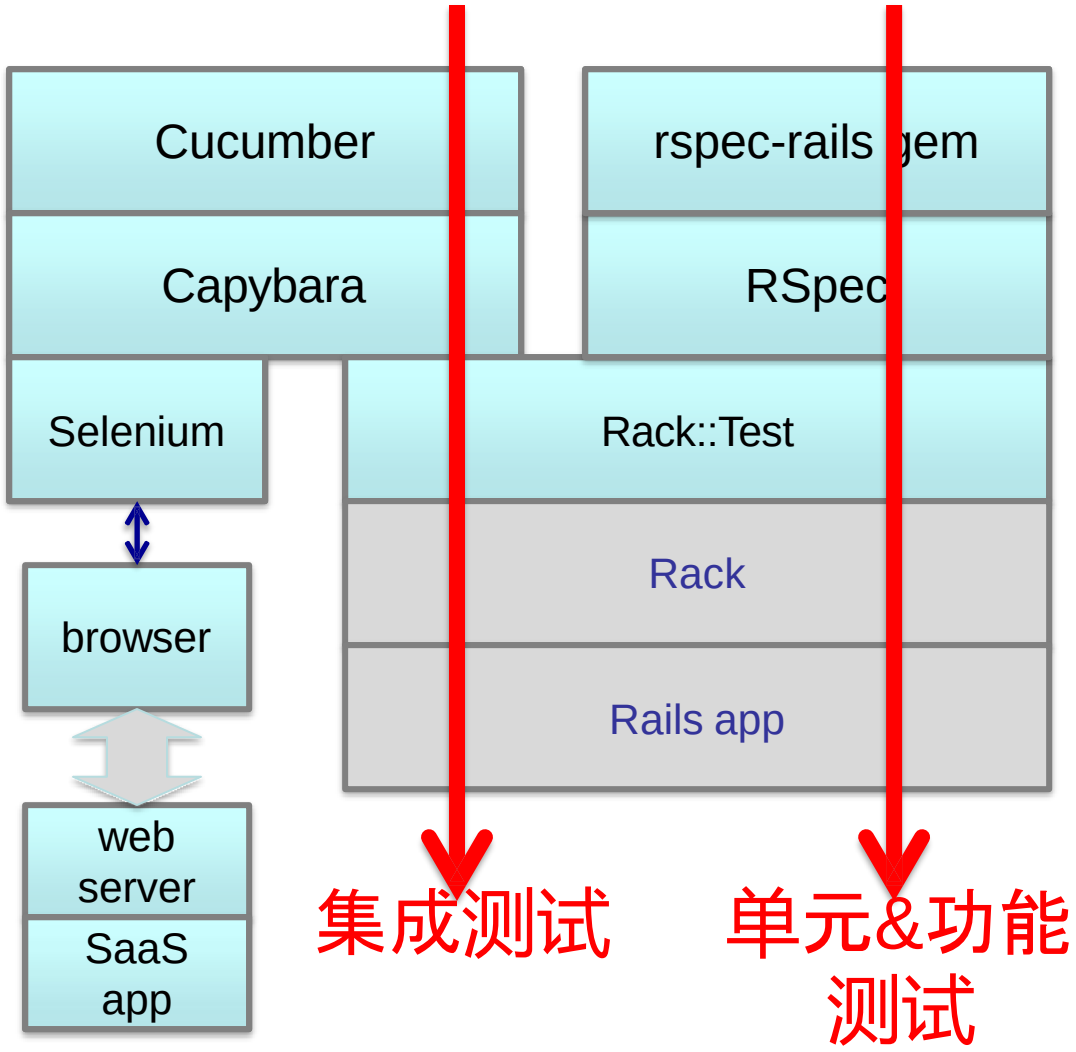
- 行为驱动的设计 （BDD）
  - 开发用户故事(希望拥有的功能特征)来描述应用程序将如何工作
  - 通过Cucumber，用户故事成为验收测试和集成测试用例
- 测试驱动的开发 （TDD）
  - 新故事的步骤定义对应着可能需要编写的新代码
  - TDD：在代码本身之前，先为代码编写单元和功能测试
  - 也就是说：为你希望拥有的代码编写测试

Cucumber与Rspec（BDD与TDD）

- Cucumber通过特征和场景描述行为(行为驱动设计)
- RSpec测试对应这些行为的各个程序模块(测试驱动开发)



测试技术栈回顾


Cucumber
rspec-rails gem
Capybara
RSpec
Selenium
Rack::Test
Rack
browser
Rails app
web server
SaaS app
集成测试
单元&功能测试

关于BDD和TDD，哪些是正确的:

a) 需求驱动实现

b) 它们只能在敏捷开发中使用

c) 他们拥抱并应对变化

- □ Only (a)
- □ Only (a) & (b)
- □ Only (a) & (c)
- □ (a), (b) and (c)

FIRST, TDD,开始使用RSpec

(Engineering Software as a Service §8.2)

单元测试应该FIRST

- Fast-快：快速运行(测试的子集)(因为将一直运行它们)
- Independent-独立：没有测试依赖于其他测试，因此可以以任何顺序运行任何子集
- Repeatable-可重复：运行N次，得到相同的结果(以帮助隔离bug并启用自动化)
- Self-checking-自检查：测试可以自动检测是否通过(无需人工检查输出)
- Timely-及时：与被测试的代码同时编写(使用TDD，先编写!)

RSpec，用于测试的领域特定语言（Domain Specific Language）

- DSL：以牺牲通用性为代价，简化一项任务的小型编程语言
  - 课程中到目前为止的例子：数据库迁移、正则表达式、SQL
- Rspec测试被称为 *specs* 或者例子

http://pastebin.com/LKTK36Pb

- 运行一个文件中的测试：**`rspec`** *filename*
  - Red 失败，Green 通过，Yellow 挂起
- *更好的方式：运行* **`autotest`**

```ruby
require 'dessert'
require 'debugger'
describe Dessert do
  describe 'cake' do
    subject { Dessert.new('cake', 400) }
    its(:calories) { should == 400 }
    its(:name)     { should == 'cake' }
    it { should be_delicious }
    it { should_not be_healthy }
  end
  describe 'apple' do
    subject { Dessert.new('apple', 75) }
    it { should be_delicious }
    it { should be_healthy }
  end
  describe 'can set' do
    before(:each) { @dessert = Dessert.new('xxx', 0) }
    it 'calories' do
      @dessert.calories = 80
      @dessert.calories.should == 80
    end
    it 'name' do
      @dessert.name = 'ice cream'
      @dessert.name.should == 'ice cream'
    end
```

RSpec基础示例

```
x = Math.sqrt(9)
x.should == 3
(sqrt(9)).should == 3
expect { sqrt(9) }.to == 3

x.odd?.should be_true
x.should be_odd

m = Movie.new(:rating => 'R')
m.should_not be_valid
m.should be_a Movie
```

更多RSpec基础示例

```
expect { m.save! }.
 to raise_error(ActiveRecord::RecordInvalid)
m = (create a valid movie)
m.should be_valid
lambda { m.save! }.
  should change { Movie.count }.by(1)
expect(m).to be_valid
expect { m.save! }.
  to change { Movies.count }.by(1)
```

```
expect { actions }.to(assertion)
expect(object).to(assertion)
```

哪种代码可以被重复(R)且独立地(I)测试?
a)依赖随机性的代码(例如洗牌)
b)依赖于时间的代码(例如，在每个星期天的午夜运行备份)

- □ Only (a)
- □ Only (b)
- □ Both (a) and (b)
- □ Neither (a) nor (b)

RSpec on Rails

(Engineering Software as a Service §8.2)

RSpec-Rails中的特殊性

- RSpec中混合了其他方法来测试特定于Rails的内容
  - 例如. 控制器中的get, post, put ...
  - 控制器的响应对象
- Matchers来测试Rails应用程序的行为

```
response.should
    render_template("movies/index")
```

- 支持创建各种 复制品(doubles)来测试对象的方法

be_a_new		all	primarily intended for controller specs
render_template	assert_template	request / controller / view	use with expect(response).to
redirect_to	assert_redirect	request / controller	use with expect(response).to
route_to	assert_recognizes	routing / controller	use with expect(...).to route_to
be_routable		routing / controller	use with expect(...).not_to be_routable
have_http_status		request / controller / feature	
match_array		all	for comparing arrays of ActiveRecord objects
have_been_enqueued		all	requires config: ActiveJob::Base.queue_adapter = :test
have_enqueued_job		all	requires config: ActiveJob::Base.queue_adapter = :test

例子: 调用 TMDb

- RottenPotatoes新功能: 使用来自TMDb的信息添加电影(vs. 手工输入)
- 用户故事的步骤应该如何描述?

```
When I fill in "Search Terms" with "Inception"
 And I press "Search TMDb"
Then I should be on the RottenPotatoes homepage
...
```

回想“Rails 烹调法 #2”:

增加新功能 ==

新路由 + 新控制器方法 + 新视图

你希望拥有的代码

接收搜索表单的控制器方法应该做什么?

1. 它应该调用一个（模型）方法，在TMDb中搜索指定的电影
2. 如果找到：它应该选择(新的)“搜索结果”视图来显示匹配
3. 如果没有找到匹配：它应该*重定向*到带有“未找到”消息的RottenPotatoes主页

http://pastebin.com/kJxjwSF6

```
1.require 'spec_helper'
2.
3.describe MoviesController do
4.  describe 'searching TMDb' do
5.    it 'should call the model method that
6.        performs TMDb search'
7.    it 'should select the Search Results
8.        template for rendering'
9.    it 'should make the TMDb search results
10.      available to that template'
11.end
12.end
```

访问TMDb来搜索某个电影的**方法**应该是:

- Movie模型的一个类方法
- Movie模型的一个实例方法
- 一个控制器方法
- 一个辅助(Helper)方法

TDD周期：红–绿–重构

(Engineering Software as a Service §8.3)

测试优先(Test First)的开发

- 考虑 代码应该完成的一件事（功能）
- 在*失败的*测试中记录/捕捉这种想法
- 编写尽可能简单的代码使测试通过
- 重构:与其他测试一起来提炼共性，改进代码
- 继续完成代码应该做的下一件事（功能）

红 - 绿 - 重构

以“始终拥有工作代码”为目标

如果有影响测试的依赖项，
如何“隔离地”测试它?

你希望拥有的代码

接收搜索表单的控制器方法应该做什么?

1. 它应该调用一个方法，在TMDb中搜索指定的电影
2. 如果找到：它应该选择(新的)“搜索结果”视图来显示匹配
3. 如果没有找到匹配：它应该*重定向*到带有“未找到”消息的RottenPotatoes主页

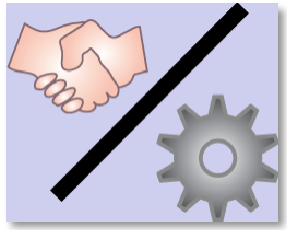
http://pastebin.com/kJxjwSF6

面向控制器动作的TDD：设置

- 在**config/routes.rb**中增加一个路由

```
#发送“Search TMDb”表单的路由
post '/movies/search_tmdb‘
```

  - 约定优先原则将会将其映射到 MoviesController#search_tmdb



- 创建一个空视图：

```
touch app/views/movies/search_tmdb.html.haml
```

- 用空方法替换**movies_controller.rb**中的假“硬连接”方法：

```
def search_tmdb
end
```

模型方法是什么样?

- 调用TMDb是模型Movie的职责……但是还没有模型方法来做这件事!
- 没问题.....我们“伪造”对“我们希望拥有的代码”的调用 (*“CWWWH”*), `Movie.find_in_tmdb`
- 行动步骤:
  - 模拟将***搜索表单***发布(Post)到控制器动作（方法）
  - 检查控制器动作(方法)是否使用提交表单中的数据来调用 `Movie.find_in_tmdb`.
  - 测试将失败(红色)，因为(空)控制器方法没有调用 `Movie.find_in_tmdb`.
  - 修复/填充控制器方法使得测试变绿.

http://pastebin.com/zKnwphQZ

测试 控制器的动作

```
1.require 'spec_helper'
2.
3.describe MoviesController do
4.  describe 'searching TMDb' do
5.    it 'should call the model method that
      performs TMDb search' do
6.      Movie.should_receive(:find_in_tmdb)
          .  with ('hardware')
7.      post :search_tmdb, {:search_terms =>
         'hardware'}
8.     end
9.  end
10.end
```

关于should_receive下面哪一个是假的?

- 它为一个还不存在的真正方法提供了一个替身
- 它将覆盖真正的方法，即使它确实存在
- 它可以在应该发出调用的代码之前或之后产生
- 它利用Ruby的开放类和元编程机制在测试时“拦截”方法调用

接缝(Seams)

(Engineering Software as a Service §8.3)

接缝(Seam)

- 可以在不改变源代码的情况下改变应用程序行为的位置.
  (Michael Feathers, *Working Effectively With Legacy Code*)
- 对测试有用：将某些代码的行为与它所依赖的其他代码隔离开来.
- `should_receive`使用Ruby的开放类创建一个seam，用于将控制器动作与真正`Movie.find_in_tmdb`的行为(可能有bug或缺失)隔离开来
- Rspec在每次测试之后重置所有Mock & stub (保持测试独立)

如何使这个spec执行变绿?

- ***期望(Expectation)*** 说控制器动作应该调用 `Movie.find_in_tmdb`
- 那么，让我们就调用它!

http://pastebin.com/DxzFURiu

```
1.def search_tmdb
2.  Movie.find_in_tmdb(params[:search_terms])
3.end
```

该spec已经驱使创建了控制器方法，从而使得该测试可以通过。

- 但是 `find_in_tmdb` 不应该返回什么东西吗?

最终，我们将不得不编写一个真正的 find_in_tmdb。当这种情况发生时，我们应该：

- 将我们测试中的 should_receive 替换为对真正的 find_in_tmdb的调用
- 确保真正的find_in_tmdb的API与 should_receive使用的假API匹配
- 在spec中保留should_receive 接缝，但如果有必要，更改spec以匹配真正的find_in_tmdb的API
- 完全删除该spec(测试用例)，因为它不再真正测试任何东西

期望(Expectation)

(Engineering Software as a Service §8.4)

发展方向:“从外缘向中心”的开发

- 重点: 编写**期望**，从而驱动 控制器方法的开发
  - 发现: 必须与 模型方法 协作
  - 递归地使用由外向内开发: 构造此测试中模型方法的存根(Stub)，以后再编写它
- 核心思想:摆脱被测方法与其合作者之间的依赖关系
- 关键概念: 缝合(seam)—你可以在不编辑代码的情况下影响应用程序行为之处

你希望拥有的代码

接收**搜索表单**的控制器方法应该做什么?

1. 它应该调用一个方法，在TMDb中搜索指定的电影
2. 如果找到：它应该选择(新的)“搜索结果”视图来显示匹配者
3. 如果没有找到匹配：它应该*重定向*到带有“未找到”消息的RottenPotatoes主页

“它应该选择(新的)“搜索结果”视图来显示匹配者”

- 2个 specs (Rspec的测试):

1. 它应该(should)决定呈现搜索结果
   - 更重要的是，可以根据结果呈现不同的视图
2. 它应该(should)使匹配的列表可用于该视图

- 基本的期望(*expectation*)结构:

```
obj.should   match-condition

  expect(obj).to match-condition
```

  - 很多内置的匹配器(matcher)，或者自定义的

Should & Should-not

- 匹配器(matcher)对*should*的接收者进行测试

`count.should == 5`	count.should.==(5) 的语法糖
`5.should(be.<(7))`	be创建一个用于测试谓词表达式的lambda
`5.should be < 7`	允许的语法糖
`5.should be_odd`	使用method_missing调用odd? 作用在5上
`result.should include(elt)`	调用 #include?, 通常是由Enumerable处理的
`republican.should cooperate_with(democrat)`	调用程序员定制的matcher #cooperate_with (可能失败)

```
response.should render_template('search_tmdb')
```

检查渲染

- 在 `post :search_tmdb`之后，`response()` 方法返回控制器的响应对象
- `render_template`匹配器用于检查绑定于某控制器方法的视图

http://pastebin.com/C2x13z8M

- 注意，这个视图必须存在！
  - `post :search_tmdb`将驱动完成整个MVC流程，包括渲染视图
    - 因此，针对控制器的specs可被视为*功能测试*

```
1.require 'spec_helper’

2.describe MoviesController do
3. describe 'searching TMDb' do
4.it 'should call the model method that performs TMDb
search' do
5.    Movie.should_receive(:find_in_tmdb).with('hardware')
6.      post :search_tmdb, {:search_terms => 'hardware'}
7.    end
8.    it 'should select the Search Results template for
rendering' do
9.      Movie.stub(:find_in_tmdb)
10.      post :search_tmdb, {:search_terms => 'hardware'}
11.      response.should render_template('search_tmdb')
12.    end
13.end
14.end
```

我们所知的测试机制

```
obj.should_receive(a).with(b)
```

```
obj.should match-condition
```

RSpec对Rails特定的扩展:

```
response()–receiver（被测试对象）
render_template() --matcher
```

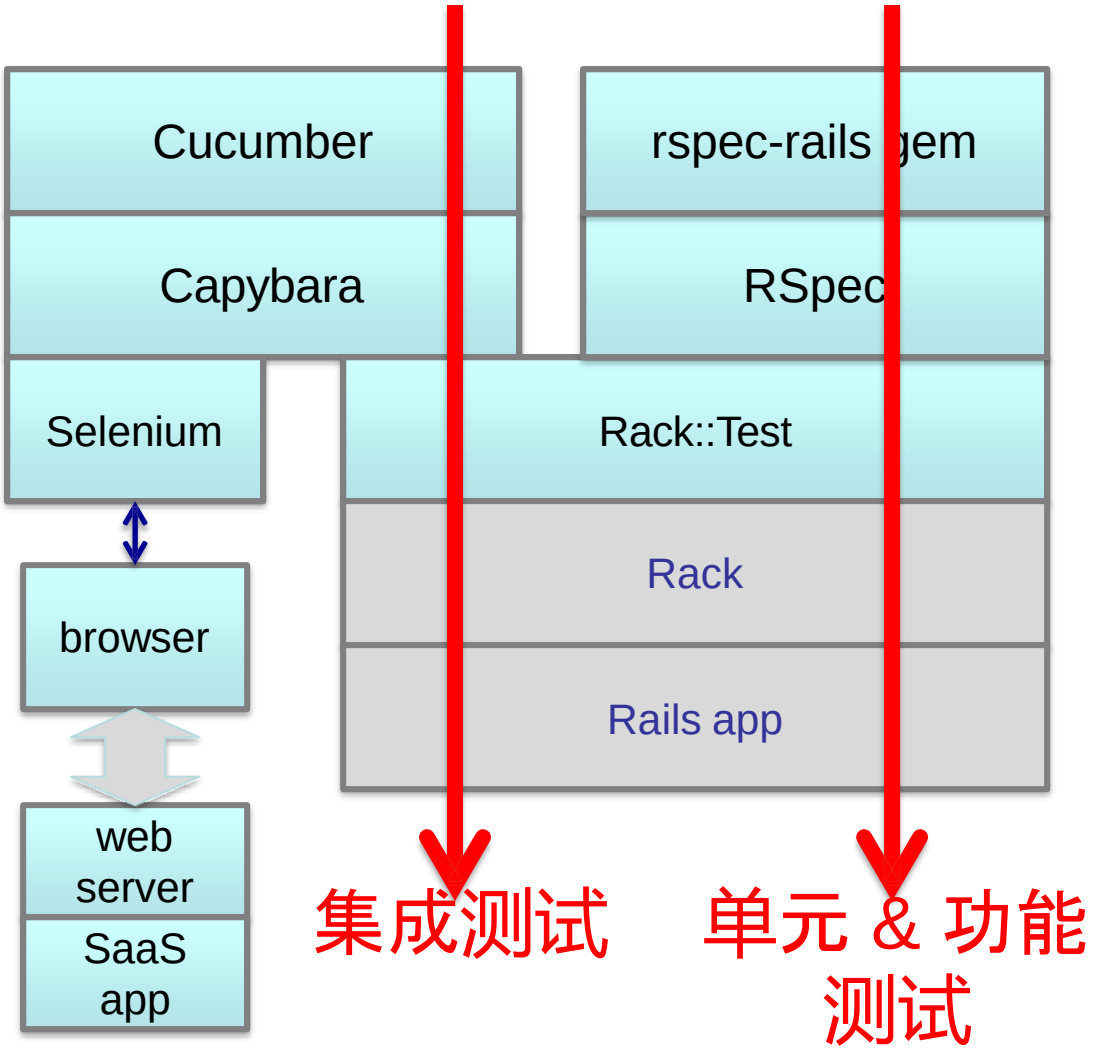
下面哪个不是should和should_not的正确用法?

- ☐ `result.should_not be_empty`
- ☐ `5.should be <=> result`
- ☐ `result.should_not match /^D'oh!$/`
- ☐ 以上都是有效的使用

模拟(Mock)和存根(Stub)

(Engineering Software as a Service §8.4)

测试技术栈回顾


Cucumber
Capybara
Selenium
browser
web server
SaaS app
rspec-rails gem
RSpec
Rack::Test
Rack
Rails app
集成测试
单元 & 功能测试

你希望拥有的代码

接收**搜索表单**的控制器方法应该做什么?

1. 它应该调用一个方法，在TMDb中搜索指定的电影 (已完成)
2. 如果找到匹配: 它应该让搜索结果对模板可用
    1. 它应该(should)决定渲染搜索结果 (已完成)
    2. 它应该(should)使匹配的列表可用于该模板

它应该(should)使匹配的列表可用于该模板

- 另外一个Rspec-Rails扩展物: `assigns()`
  - 传递命名控制器实例变量的符号
  - 返回控制器赋给变量的值
- 我们当前的代码没有设置任何实例变量:

缺实例变量

http://pastebin.com/DxzFURiu

```
1.def search_tmdb
2.   Movie.find_in_tmdb(params[:search_terms])
3.end
```

- TCWWWH: `@movies`中的匹配列表
  - 见下页

http://pastebin.com/4W08wL0X

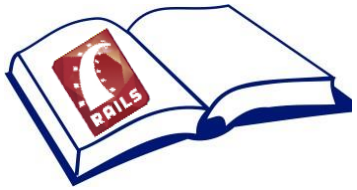
```
1.require 'spec_helper'
2.
3.describe MoviesController do
4.  describe 'searching TMDb' do
5.    it 'should make the TMDb search results available
to that template'
6.      fake_results = [mock('Movie'), mock('Movie')]
7.      Movie.stub(:find_in_tmdb).and_return(fake_results
)
8.      post :search_tmdb, {:search_terms => 'hardware'}
9.      # look for controller method to assign @movies
10.       assigns(:movies).should == fake_results
11.    end
12.end
13.end
```

两个新Seam概念

- 存根-stub
  - 类似should_receive, 但没有期望(expectation)
  - and_return可选地控制返回值
  - `Movie.stub(:find_in_tmdb).and_return(fake_results)`
- 模拟-mock: “特技替身”对象, 经常用于行为验证(让某方法被调用)
  - 在其上为单个方法构建存根:

```
m=mock('movie1',:title=>'Rambo')
```

每个seam仅为测试中的某些特定行为提供足够的支撑



RSpec烹调法 #1

- 每个spec应该只测试一种行为
- 根据需要使用接缝(seam)来隔离该行为
- 确定用什么类型期望(expectation)来检查行为
- 编写测试，并确保测试失败的原因是正当的
- 添加代码直到测试变绿
- 寻找机会重构/美化代码

我们所知的测试机制

```
obj.should_receive(a).with(b).and_return(c)
obj.stub(a).and_return(b)

d = mock('impostor')
```

可选!

```
obj.should match-condition
```

RSpec对Rails特定的扩展:

```
assigns(:instance_var)
response()
render_template()
```

should_receive 组合了 _____ 和 _____ 而 stub 只是_____.

- 一个 mock 和一个期望; 一个mock
- 一个mock 和一个期望; 一个期望
- 一个seam 和一个期望; 一个期望
- 一个seam 和一个期望; 一个seam

装置(Fixture)和工厂(Factory)

(Engineering Software as a Service §8.5)

当你需要真正的东西时

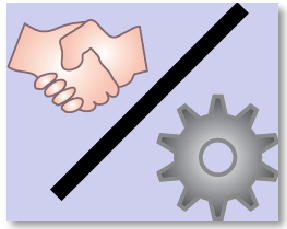
从哪里获得一个真实的对象？ http://pastebin.com/N3s1A193

- 装置：静态地将一些已知数据预加载到数据库表中
- 工厂：只创建每次测试所需的内容

```
1.fake_movie = mock('Movie')
2.fake_movie.stub(:title).and_return('Casablanca')
3.fake_movie.stub(:rating).and_return('PG')
4.fake_movie.name_with_rating.should == 'Casablanca (PG)'
```

装置

- 在每次测试之前，数据库需擦除和重新载入
  - 在describe开始处加上fixtures : movies
  - spec/fixtures/movies.yml 是Movies 将被加载进movies表中



- 赞成/使用的原因
  - 真正的静态数据，例如、永远不变的配置信息
  - 很容易在一个地方看到所有测试数据
- 反对/不使用的原因
  - 可能会引入对装置数据的依赖

工厂

- 根据测试需要，设置“helper”来快速创建带有默认属性的对象
- 例子：FactoryGirl gem
- 赞成/使用：
  - 保持测试的独立性：不受它们不关心对象的影响
- 反对/不使用的原因：
  - 复杂的关系可能很难建立（但可能表明代码耦合过紧!）

http://pastebin.com/bzvKG0VB

假设对RottenPotatoes应用，下列哪一种数据(如果有的话)不应该设置为 **装置**?

- 电影及其评级
- TMDb API key
- 应用程序的时区设置
- 所有这些都可以设置为fixture

针对*模型*的TDD & 构建Internet存根

(Engineering Software as a Service §8.6–8.7)

显式与隐式需求

- 调用`find_in_tmdb`应该使用“标题关键字”参数（TmdbRuby gem）
  - 如果我们没有gem: 它应该直接向远程TMDb站点提交一个RESTful URI
- 如果TmdbRuby gem发出错误消息怎么办?
  - API key无效
  - API key没有提供
- 使用 *context & describe* 划分测试

http://pastebin.com/ELQfC8Je

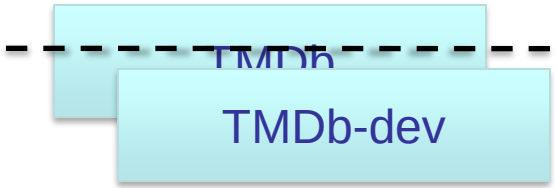
```ruby
describe Movie do
  describe 'searching Tmdb by keyword' do
    context 'with valid key' do
      it 'should call Tmdb with title keywords' do
        TmdbMovie.should_receive(:find).
          with(hash_including :title => 'Inception')
        Movie.find_in_tmdb('Inception')
      end
    end
    context 'with invalid key' do
      it 'should raise InvalidKeyError if key not given' do
        Movie.stub(:api_key).and_return('')
        lambda { Movie.find_in_tmdb('Inception') }.
          should raise_error(Movie::InvalidKeyError)
      end
      it 'should raise InvalidKeyError if key is bad' do
        TmdbMovie.stub(:find).
          and_raise(RuntimeError.new('API returned code
                404'))
        lambda { Movie.find_in_tmdb('Inception') }.
          should raise_error(Movie::InvalidKeyError)
      end
    end
```

回顾

- 隐式需求源于显式需求
  - 通过阅读docs/specs
  - 作为设计class的副产品
- 我们使用了两种不同的stub方法
  - 情况1：我们知道TMDb gem会立即抛出错误
  - 情况2：需要完全阻止gem与TMDb联系
- `context` & `describe` 将类似测试组织起来
  - 在书中：使用`before(:each)`设置适用于整个测试组的通用先决条件

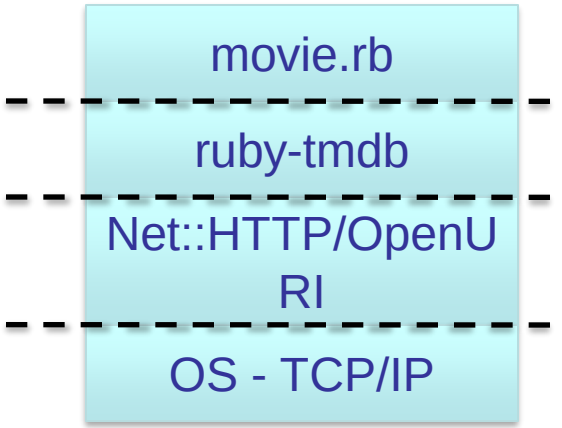
SOA架构中在哪里构造存根(stub)?


```
Movie.stub(:find_in_tmdb).
  and_return(...)
TmdbMovie.stub(:find).with(...).
  and_return(...)
Net::HTTP.stub(:get).
  with(...).
  and_return(...)
```



- 解析 人为发出的请求
- 返回封装的值
- *(例如使用* `FakeWeb` *gem)*

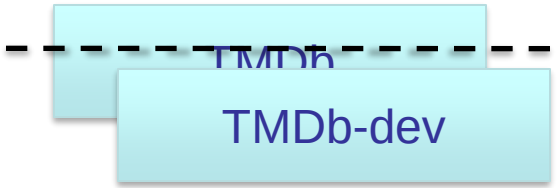
SOA架构中在哪里构造存根(stub)?




```
Movie.stub(:find_in_tmdb).
  and_return(...)
TmdbMovie.stub(:find).with(...).
  and_return(...)
```

经验法则

- 对于单元测试，就近建存根，以最大限度地隔离待测类(F-快速，I-独立)
- 对于集成测试，远一点建存根，测试尽可能多的接口



- 解析 人为发出的请求
- 返回封装的值
- *(例如使用 FakeWeb gem)*

我们所知的测试机制

```
obj.should_receive(a).with(b).and_return(c)
                   .with(hash_including 'k'=>'v')
obj.stub(a).and_raise(SomeClass::SomeError)

d = mock('impostor')

{ action }.should raise_error(Some::Error)
describe, context
```

RSpec对Rails特定的扩展:

```
assigns(:instance_var)
response()
render_template()
```

测试覆盖，
单元测试 vs. 集成测试

(Engineering Software as a Service §8.8)

多少测试才足够?

- 坏消息: “直到发布时间”
- 好一点: (测试行数)/(代码行数)
  - 1.2x - 1.5x 不是不合理
  - 对于生产系统来说通常要高得多
- 更好的问题是: “我的测试有多彻底?”
  - 形式化方法
  - 覆盖度量
  - 我们关注的是后者，尽管前者越来越受到关注

度量覆盖—基本知识

```
class MyClass
 def foo(x,y,z)
  if x
   if (y && z) then bar(0) end
  else
   bar(1)
  end
 end
 def bar(x) ; @w = x ; end
end
```

- S0: 方法覆盖
- S1: 调用覆盖
- C0: 语句覆盖
  - Ruby SimpleCov gem
- C1: 分支覆盖
- C1+判定覆盖：条件中的每个子表达式
- C2: 路径覆盖（困难, 价值如何有很大分歧）

测试的种类

- 单元
  (一个方法/类)
  - 功能 或 模块
    (一些方法/类)
- 集成/系统

运行快
高分辨率　高覆盖率
很多mock;
不测试接口

少mock;
测试接口

运行时间长　低覆盖率
低分辨率

极端

× “我踢了一下轮胎，就工作了”

× “在100%覆盖和变绿之前不要发布”

使用覆盖率来标识代码中未测试或测试不足的部分

× “关注单元测试，它们更彻底”

× “专注于集成测试，它们更现实”

两种方法都能找到对方没有发现的漏洞

对于TDD，以下哪个建议是糟糕的?

- 在单元测试中就近、经常性地创建Mock&Stub
- 单元测试以高覆盖率为目标
- 有时在集成测试中使用stubs & mocks是OK的
- 单元测试比集成测试能带来对系统正确性有更高的信心

其他测试的概念；
测试与调试

(Engineering Software as a Service §8.9, 8.12)

可能听到的其他测试术语

- **变异测试**(Mutation testing)：如果在代码中引入故意的错误，是否有一些测试失败?
- **模糊测试**(Fuzz Testing)：1万只猴子向你的代码扔随机输入
  - 发现了MS~20% bug,崩溃了~25%的Unix应用工具
  - 以应用程序不应该被使用的方式来测试它
- **DU-覆盖率**：每一对<define x/use x>执行了吗?
- **黑盒 vs. 白盒测试**

TDD vs.传统的调试

Conventional	TDD
写10行代码，运行，遇上bug: 在调试器中中断	
在重复运行时插入printf以打印变量	
在调试器中停止，调整/设置变量来控制代码路径	
该死，我以为我肯定修好了，现在我必须重新做一遍	

•教训 1: TDD使用与传统调试相同的技能和技术——但更高效(FIRST)

• 教训 2: 在编写代码之前编写测试需要花费更多的时间，但总体上花费的时间通常更少

TDD 小结

- 红 - 绿 - 重构，并始终有工作代码
- 使用接缝seam，一次测试一种行为
- 使用it“占位符”或pending记录你知道需要的测试
- 阅读和理解测试覆盖率报告
- “纵深防御”：不要过于依赖任何一种测试

关于测试，哪个说法是错误的?

- 即使100%的测试覆盖率也不能保证没有bug
- 如果可以在调试器中触发引起错误的条件，那么你可以在测试中捕获它
- 测试消除了使用调试器的需要
- 当更改代码时，也需要更改测试

软件测试的P&D视角

(Engineering Software as a Service §8.10)

P&D 中的测试是怎样的?

- BDD/TDD在编写代码之前编写测试
  - P&D开发人员什么时候编写测试?
- BDD/TDD从用户故事开始
  - P&D开发者从哪里开始?
- BDD/TDD开发人员编写测试和代码
  - P&D是使用相同的人还是不同的人进行测试和编码?
- 测试计划和测试文档是什么样子的?

P&D项目经理

- P&D 依赖于项目经理
- 编制项目管理计划文件
- 创建软件需求规格说明(SRS)
  - 可以是数100页
  - 遵循IEEE标准
- 必须编制测试计划
  - 遵循IEEE标准

P&D测试方法

- 经理将SRS划分为编程单元
- 开发人员编写单元代码
- 开发人员执行单元测试
- 独立的质量保证(QA)团队进行更高级别的测试:
  - 模块，集成，系统，验收测试

3个QA集成选项

1. 自顶向下集成
   - 从依赖关系图的顶部开始
   - 高级功能(UI)很快就能工作了
   - 构建许多存根Stub让应用程序“工作”
2. 自底向上集成
   - 从依赖关系图的底部开始
   - 没有存根，把所有的集成在一个模块中
   - 不能看到应用程序工作，直到所有代码编写和集成完成

3个QA集成选项

3. 三明治集成
   - 两全其美?
   - 通过自底向上集成一些单元来减少存根-stub构造
   - 通过自顶向下集成一些单元，尝试让UI可操作

QA团队测试

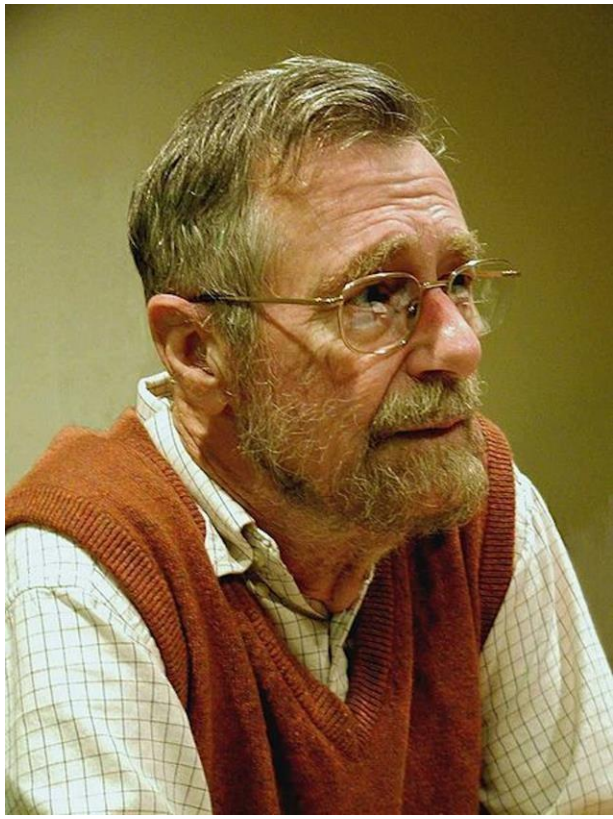
- 接下来QA团队进行系统测试
  - 完整的应用程序应该可以运行
  - 测试非功能需求(性能)+功能需求(SRS中的特性)
- 什么时候进行系统测试?
  - 机构的政策
    - 例如，测试覆盖级别(所有语句)
    - 例如，所有测试要输入好的和坏的数据

- 最后一步：客户或用户验收测试(UAT)——
  —　确认 vs. 验证

测试的局限

- 程序测试可以用来显示bug的存在，但永远不能证实它们的不存在！
  - Edsger W. Dijkstra

（获得1972年图灵奖，以表彰其对开发编程语言的重要贡献）



形式化方法

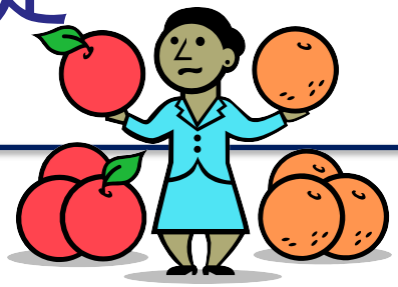
- 从形式规格说明开始，并证明程序行为符合规格说明。

1. 人类做证明
2. 计算机通过自动定理证明
   - 使用逻辑公理+推理，从头开始产生证明
3. 计算机通过模型检查
   - 通过穷尽搜索系统在执行期间可能进入的所有可能状态来验证选定的性质

形式化方法

- 计算复杂度高，针对以下情况使用
  - 小而固定的功能
  - 昂贵的维修，很难测试
  - 例如、网络协议，安全关键软件
- 更大:
  - OS内核，10K LOC，$500/LOC
  - NASA软件 $80/LOC
- 本课程：快速变化的软件(SaaS)，易于修复，易于测试 => 不使用形式化方法

软件测试：P&D vs. 敏捷 (Fig. 8.26)



Tasks	*In Plan and Document*	*In Agile*
Test Plan and Documentation	Software Test Documentation such as IEEE Standard 829-2008	User stories
Order of Coding and Testing	1. Code units 2. Unit test 3. Module test 4. Integration test 5. System test 6. Acceptance test	1. Acceptance test 2. Integration test 3. Module test 4. Unit test 5. Code units
Testers	Developers for unit tests; QA testers for module, integration, system, and acceptance tests	Developers
When Testing Stops	Company policy (e.g., statement coverage, happy and sad user inputs)	All tests pass (green)

哪一项关于测试的陈述是错误的?

- 形式化方法是昂贵的，但值得验证重要的应用程序
- P&D开发人员在编写测试之前先编写代码，而敏捷开发人员反之亦然
- 敏捷开发人员执行模块、集成、系统和验收测试;而P&D开发人员不做
- P&D中的**三明治集成**旨在减少构造存根-stub的工作量，同时尝试尽早获得通用性功能