

TECHNISCHE FACHHOCHSCHULE BERLIN

Fachbereich VI: Medien und Informatik

Studiengang: Medien-Informatik

TEST DRIVEN WEB DEVELOPMENT WITH RUBY ON RAILS

Bachelorarbeit

zur Erlangung des Grades eines Bachelors

eingereicht von: .Michael Grosser

(Matrikel-Nr. 741163)

am: 30.04.2009

Erstgutachter: Prof. Dr. Edlich

Zweitgutachterin: Prof. Dr. Merceron

1. CONTENT

1.	Content.....	2
2.	Overview of Technologies used.....	3
2.1	Ruby.....	3
2.2	Web application.....	4
2.3	Ruby on Rails	5
3.	Introduction.....	6
3.1	Abstract	6
3.2	Problem background	6
3.3	Main problem	6
3.4	Purpose.....	8
3.5	Scope	8
3.6	Conclusion	8
4.	Testing Basics	9
4.1	Introduction.....	9
4.2	Getting started	9
4.3	Guidelines.....	12
4.4	Fixtures	13
4.5	Develop without opening a browser	13
4.6	Testing views	14
4.7	Conclusion	14
5.	Testing Tools.....	15
5.1	Introduction.....	15
5.2	Installation Tips.....	15
5.3	Tools	17
5.4	RSpec	25
5.5	Javascript	30
5.6	Helpers and Tasks	35
5.7	Plugin Recourses.....	36
5.8	Conclusion	36
6.	Example TDD	37
6.1	Introduction.....	37
6.2	Basic User tests.....	37
6.3	Basic controller test.....	38
6.4	Converting to RSpec	39
6.5	Splitting the controller.....	39
6.6	Starting from scratch	43
6.7	Conclusion	45
7.	Conclusion	46
8.	List of figures	47
9.	Sources	48

2. OVERVIEW OF TECHNOLOGIES USED

2.1 RUBY

Ruby emerged on the software landscape in late 1995. Initially a part-time project of its creator Yukihiro "Matz" Matsumoto Ruby became successful through its unique focus on readability, simplicity and openness.

Often [...] computer engineers, focus on the machines. They think, "By doing this, the machine will run [...] more effectively[...]" But in fact we need to focus on humans[...]. We are the masters. They are the slaves. --- Matz

Readability is the feature most people see immediately when getting in contact with Ruby code.

A simple task like printing 'hello' 5 times can be a cryptic text only decipherable by adept programmers like this Java block (left) or just understandable and simple like ruby (right).

```
for(int i=0;i<6;i++){System.out.println("Hello");}           print 5 * "Hallo"
```

This level of readability can only be achieved by making many of the established code conventions optional. No more; to terminate a statement, () for method arguments, return for return values, freeing the "if" and utilizing punctuation character for method names.

```
delete record if record.new?  
def say_hello  
  "hello"  
end
```

Openness and modularity are a great source of power in Ruby. All classes can be changed at runtime, allowing for context-specific methods on base classes like `String`. Functionality can be placed in modules and be reused without changing inheritance chains, which frees programs from deep and fragile inheritance trees that are common in most other programming languages. For example a Singleton class can be as simple as this:

```
class ResourceManager  
  include Singleton  
  ...  
end  
ResourceManager.instance.find "example"
```

Simplicity is another virtue of Ruby. The programmer is freed from the complexity of choice between logically equivalent classes like `int`, `BigInt`, `double`, `float`, `Vector` and `LinkedHashSetMapList` by only having to choose between `Integer`, `Float`, `String`, `Array` and `Hash`, which simplifies writing and learning Ruby. A golden rule in ruby is "least surprise", e.g. the normally failure-attracting case (also known as `switch`) statement does not fall through. The rule also implies that a method should not perform actions which cannot be derived from its name. This can be observed on `String`, where modifying operations all use the "!", to be distinguishable from their harmless counterparts that only return a modified copy e.g. `String.reverse!` and `String.reverse`.

Ruby is a dynamic scripting language, which allows assigning a value to any variables, without restriction to a specific type. A complete program can be as short as a single method call and does not need to be compiled, making it easy to start programming and to write small scripts.

The language has had a small community of enthusiasts until in 2003 Ruby on Rails arrived. "Rails" is a web-development framework which uses many techniques and architectures that would not be possible in most other languages. Its new simplicity, ease of use and reusability has attracted countless programmers. Since the release of Rails the number of Rubyists has strongly increased and the demand for Ruby or Rails developers has more than doubled every year since 2005 (Indeed, 2009).

2.2 WEB APPLICATION

In contrast to a desktop program a typical web-application is restful. Users do not interact with a live application, but only see a representation of its current state in their browser and can send requests to the server to see another state or send data so that the current state changes.

This architecture yields many unique challenges, like knowing which user has access to a certain part of the application, handling concurrent updates without compromising data integrity, timely responses to users from all over the world, working without client-side storage, varying security limitations from different clients and many more.

Working with these limitations can be hard but nevertheless web applications manage to claim a huge market share, because they allow millions to use an application without prior installation, enables them to work together, to receive instant software updates, to share their data and use their data from anywhere in the world.

A standard HTML based web application uses 3 server-side layers; a database for data-storage and retrieval, a controller that receives and responds to user requests and a view that represents the current application state as HTML.

Traditional thick client



Web based thin client

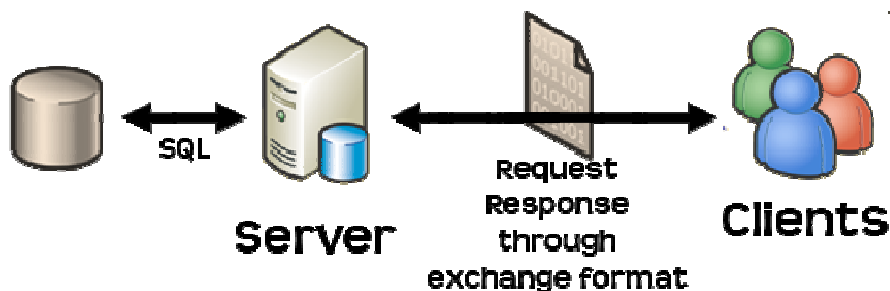


Figure 1: Comparisson of thick and thin client

2.3 RUBY ON RAILS

Rails was created by David Heinemeier Hansson in 2003, on his quest to make a simple, extendable web-application framework, that makes building fast and easy as long as one obeys commonly acknowledged best practices.

Rails is built as Model-View-Controller, where the Model is an Active-Record (Fowler, Patterns of Enterprise Application Architecture, 2002, p. 37) , the Controller translates a browsers request to an action and then displays the result of this action in a View.

By default Rails is split in 3 environments, namely production, development and test. In production everything is optimized for speed and reliability, whereas development focuses on helpful error reporting and a direct change-result feedback. Testing behaves like development but has all dangerous action turned off, it will not send real emails or only use a mocked payment system.

Conventions are what makes development so dry, it is not necessary to specify where model file lies, or which controller is called when a user visits `http://website.com/movies/1` . Not even which fields models uses, since this is all automatically inferred. For example if the database has a column named `email` in the users table, then the User model, which lives in `/app/models/user.rb`, will have an email field, no configuration or setup required.

Since these conventions define how a application should be organized, many generators have been built, that obey these conventions and generate common or otherwise repetitive code. They allow users to simply get started, without having to read endless documentation.

```
script/generate scaffold car name:string model:string description:text
```

The application is complete! Cars can be created, shown, updated and deleted. The generated code helps new users to understand what they have to do when they want to build any custom logic and prevents experienced users from having to define the same standard controllers and models all over. Furthermore tests are created, so that users know how they should test and where they should test which component.

Testing is a central part of Ruby's and Rails philosophy. Rails provides the user with countless testing helpers and an organized testing structure, so that models, controllers, views, helpers, mailers, ..., can be tested with ease. As the scaffold generator shown above, all other generators build tests for code they produce.

Plugins are where most reuse can be found in the Rails world. Every piece of logic that many people use has been turned into a plugin, they are easy to create, easy to install (`script/plugin install URL`) and act as a separation of concerns since they contain all relevant logic and tests for one part of the application. Plugins are very powerful, since Ruby allows them to alter any class in the Rails ecosystem, meaning that no Rails developer must build a public API before anyone can change Rails behavior. The Rails conventions make it easy to integrate plugins since they obey the same conventions the developer uses, mostly making additional configuration obsolete.

Through those concepts and the resulting productivity and reuse, many developers have been drawn to Rails. Other web-frameworks have been heavily influenced by it (Groovy on Rails, CakePHP, ...), resulting in Rails-clones in many different programming languages. Even the enterprise world starts to notice Rails as a possible deployment platform, after large-scale websites (twitter, yellow-pages, basecamp...) have been successfully built with it.

3. INTRODUCTION

3.1 ABSTRACT

Most small to medium size web development companies employ agile methods, because they enable them to better compete in the fast-changing web-application market. These agile methods put a new focus on developer tests and automated testing, were before a testing department would write tests or interact with the finished product. At the same time web-applications are one of the most test-unfriendly environments due to their different languages, platforms and interaction principles.

In this thesis I therefore focus on the possibilities and techniques for building test-driven web-applications, with one of today's most popular web-development frameworks that brought a new focus on testing to web-development: Ruby on Rails.

3.2 PROBLEM BACKGROUND

Over the last decade monolithic development approaches like waterfall or V-Model have lost many followers to more modern, agile approaches that focus on delivering results early in the process while reducing planning overhead. A study of 1000 waterfall projects e.g. shows their failures are in 82% of the cases attributed to waterfall practices (Thomas, 2001). A similar study on 400 projects shows that for waterfall projects 10% of the developed code was actually deployed, and out of that only 20% was used (Cohen, Larson, & Ware, 2001). Whereas studies on agile projects show that they not only improve the quality of written code, but also tend to make the developer more productive (Erdogmus, Morisio, & M, 2009).

These agile methods are most often used by companies that have a single team per project because they work best when communication overhead is low. Since these projects lack intense planning they often require a lot of refactoring, which is hard to do when the code is not tested or cannot be tested automatically. Therefore a complete test suite is mandatory to keep an agile project maintainable and healthy.

When testing last, as many of the static programming methods teach, it seldom is visible with which quality tests are written, since most of the time only coverage is observed and maintainability or robustness are an afterthought. With agile approaches testing is vital for the success of the projects and unmaintainable or brittle tests are found early in the development process, since they always require rework or break on every change. Using a Test-Driven approach makes a lot of sense in this environment, since the benefits are instantly visible through reduced maintenance and faster development.

3.3 MAIN PROBLEM

3.3.1 OUTLINE

Ruby on Rails focuses on testing and enables developers to write tests for many parts of an application that are not testable with other frameworks. The default testing toolset of Rails may be larger than on other frameworks, but is still not sufficient to build a complete, automated and robust test suite. Some parts of the application cannot be tested by standard tests and others are maintenance-intensive when built only with the supplied toolset. On top of that, there is no testing manual or methodology that can be followed; only example code and structure that are not sufficient to illustrate how test-driven development should be done.

3.3.2 TESTING A WEB APPLICATION

Web development, especially HTML/Ajax based, has a long tradition of untested code, because unlike in a thick client programming environment the developers here have to handle 2 separate domains:

- Model/HTML generation (calculations and rendering) through MVC
- Browser behavior (page change/forms/redirects...)

In many web-application frameworks, developers can only test that calculations inside the model are correct (normal Unit testing) and that the rendered HTML is valid. Testing that the correct action is called upon entering a URL is mostly impractical since often Apache rewrite rules are used for this purpose, which are hard to test. For the same reason it is hard to fake a request-response cycle when one first has to set up all kinds of global variables (php: `$_GET`, `$_REQUEST`...) or build a session object. Some modern web frameworks provide ways of testing (Symfony/CakePHP..) , but for most small framework this stays a half-hearted approach, therefore automated Controller or view testing is only achievable through a remote controlled browser, which is difficult to setup and use.

Rails changes this, by allowing programmers to test between controller-actions and view-rendering. Now they can make requests, see the controller change and verify that a view works with edge-case inputs. With the help of plugins it is even possible to submit forms or click links. All these basic actions allow to thoroughly test code without opening the browser.

If we still were in web 1.0 land with 98% users on IE5, no CSS or JS, the story would end here and the web developers would live happily ever after...

3.3.3 BROWSER DIFFERENCES

While most other languages have one default compiler or platform, Javascript has many. Every browser interprets Javascript and something as simple as `{ 'a' : 1, }` will work in Firefox and Opera and will crash in Internet Explorer. While most other platforms can simulate almost any user interaction, even the most basic, like simulating a mouseover, fail with Javascript. The main problem is not that no Javascript compiler exists (there are even some that run from the command line like rhino), but that passing tests in any of them does not mean the code will run in the real, browser-world. Therefore even if Javascript syntax or execution was tested from inside Rails, it still would not be sure that it works in all browsers, leaving only the possibility to test inside real browsers.

Hence any good test has to run through a browser, which means opening a browser and going through all test pages. As far as I have seen this burden seems too hard for many developers, because most low-to-medium sized projects lack even basic Javascript tests.

Another part that is very hard to test is CSS, since it changes the graphical output and has no testable text-representation and therefore verifying that a page looks the same on all browsers can hardly be automated. Watching the same page in all browsers is a very dull task and it has to be repeated after any change. It is possible to test some basic parts of CSS, for example positions of elements, can be checked with Javascript by testing if the `offsetWidth/offsetHeight/...` of an element is correct (quircksmode.org, 2006). Since testing everything, for example if opacity works or not is nearly impossible, only basic alignment or pixel-exact positioning can be tested.

To visually test a layout in all major browsers many professionals use BrowserShots.org or a similar services, where a webpage is rendered in different browsers and the resulting screenshots are returned. Most are free as long as one is willing to wait for ca. 2 hours until the screenshots arrive. Priority processing can be bought for a monthly fee, which is worth the time saved to setup all browsers and systems needed for such a complete test.

Webhorror - ACID 2 (2005)

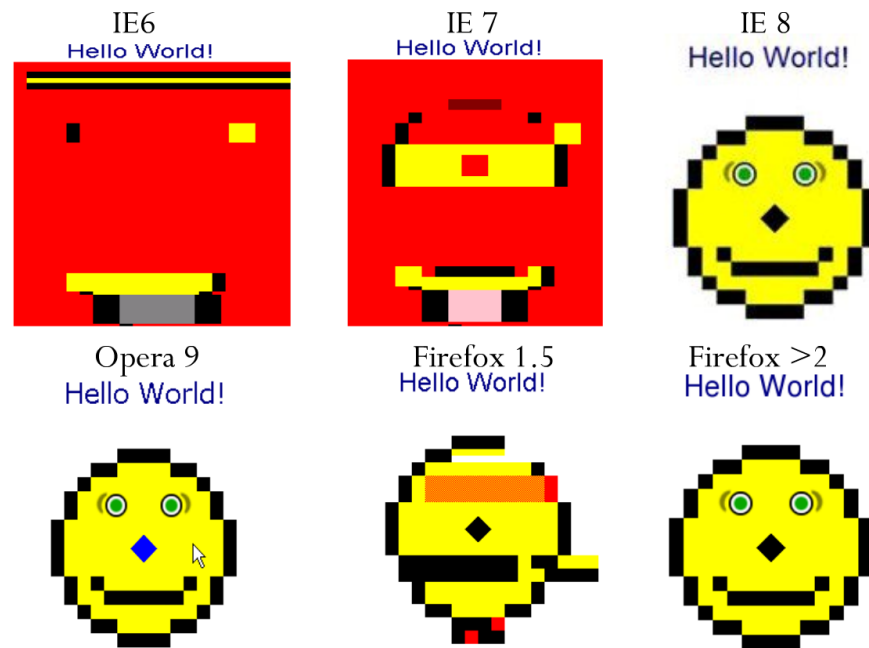


Figure 2: ACID test - same CSS, different browser

3.4 PURPOSE

The purpose of this thesis is to present different test-driven approaches of building Ruby on Rails application. For this goal a fundament of testing methods and skills must be laid, on top of which the application can be constructed by using various testing tools. Since having the right tools and knowing what is possible is very important, a main focus will be laid on which tools exist, what they are capable of and how they can work in unison.

3.5 SCOPE

The Rails ecosystem contains thousands of testing tools and frameworks, I chose to present only those that are backed by a larger community or I found useful while building applications myself.

3.6 CONCLUSION

Testing web applications bears many challenges and often is time consuming and some parts, like page layout, can only be tested manually. Rails offers a basic set of tools to make testing effective and provides tests for all code it generates, which is an advantage compared to many other web frameworks that seldom encourage their users to write tests. But these basic tools are not yet sufficient to thoroughly test a whole application, because some basic parts like e.g. Javascript are not testable with standard Rails tests. Therefore to build a complete and automated test from model through controller and views up to Javascript many more specialized tools are required.

4. TESTING BASICS

4.1 INTRODUCTION

Testing can be a waste of time when it is not done with the right techniques. Simply “writing a test” may be a good intention, but if a test does not prove any feature or breaks on any minor change, the test is worthless.

Testing first is a technique to bypass most of the problems that testing last creates. By writing tests before writing code, developers are forced to think about how the code could be made testable, which leads to loosely coupled code (McConnel, 2004, p. 118) and cannot lead to tests that prove nothing, since it is instantly visible if the test for a non-existent feature succeeds.

When writing the code to satisfy tests, just write enough to pass, not more. Writing more means writing something that is not tested, but looks as if it was tested. Finish the first idea, see it pass and then extend it with a new test case. Finally refactor, for code maintainability or performance.

When something goes wrong or an email saying “when I do this, it shows me that” comes in, it is the perfect opportunity to build a test case with this information and see it fail. And in the case that it does not fail, a new test case was build and the problem was further narrowed down. See 5.3.3 “Bug to Testcase”.

4.2 GETTING STARTED

4.2.1 BEFORE WRITING TESTS, LEARN

It is essential to not just start writing tests and hope to refactor or fix any error that occurs. Be sure that what is build will work. Understand the problem domain by reading the manual or looking at example code. After the basic idea is clear, start with a prototype (a real throw-away-and-never-look-at-again kind of prototype). Alternatively `irb` (Ruby console) or `script/console` (always use “script/console test” to not mess up the development environment) can be of great help in learning the basics of a new library.

Even after starting to test, some problems may occur, that cannot be fixed with testing alone. If it is unclear what is right or why something fails, reading and playing with the console is faster than writing countless tests.

Sample console session, after getting stuck with a failing test:

```
script/console test
Loading test environment (Rails 2.2.2)
>> r = Rating.new
=> #<Rating id: nil, r1: 0, r2: 0, r3: 0, r4: 0, r5: 0, rating: nil, movie_id: nil,
created_at: nil, updated_at: nil>
>> r.save!
ActiveRecord::StatementInvalid: Mysql::Error: #23000Column 'movie_id' cannot be
null: INSERT INTO ...
>>r.movie = Movie.new
=> #<Movie id: nil, country: nil, year: nil, duration: nil, title: nil, director:
nil,
>> r.save!
Mysql::Error: #23000Column 'movie_id' cannot be null
#Movie was assigned and there is no id, digging deeper may help...
>> r.movie.save!
ActiveRecord::RecordInvalid: validation failed: Title can't be blank
#problem found!
>> r.movie.title = 'test'
>> r.movie.save!
```

```

=> true
#success, now check if all other values are correct too
>> y r #y is a shorthand for outputting a readable(YAML) version of an object
--- &id003 !ruby/object:Rating
attributes:
  rating: 0
  movie_id: 633756704
  id: 953125646
  r1: 0
...

```

After having learned where the problem exactly lies, it is easy to see if either the testcase needs repair or if the logic was defective in the first place.

4.2.2 INTEGRATION AND UNIT TESTING

There are 2 common types of testing mentalities, integration testing (testing deep) and unit testing (testing small). Integration testing means: writing a test case that touches a lot of code and verify that all works in unison. This can work good to verify that something is working as expected, since failures would be visible in the final result. This kind of testing is often used for test last development, since testing here only serves as verification that the code runs as expected. But this approach has many drawbacks. There is no safety that everything is tested, different failures arise when changing one aspect, most failure messages are unclear thus requiring more “digging” and it is impractical for test-driven development, since everything has to be build at once.

In consequence it is better to rely on unit testing for test driven development. But unit testing can go wrong easily, some bad examples include testing more than one aspect in a single test case, testing 2 paths of execution (success and failure) at once, testing something without verifying an underlying assumption first, not naming the test cases descriptive and many more. Here is a small example to illustrate:

```

def test_new_rating_should_change_score
  assert_equal "2.83", @rating.rate!(2).rating
end

```

➔ Assumes that rate! returns itself / has a too general name

There is a border where testing something to trivial will only slow down work, but in general it is better to err on the side of making to small test, since they are easier to understand and maintain.

An example test-case refactoring:

```

def test_save
  get :edit, :id => Rating.first.id
  submit_form :rating => {:r1 => 3}

  assert_response :redirect
  assert_equal 3, Rating.first.r1
  assert_match(/success/, flash[:notice])
  assert_redirected_to :controller => 'rating', :action => 'index'
end

```

Refactoring steps to better unit tests:

- Split tests, so that one assertion is made per test case
- Move requests to a helper method, since they will be repeated
- Remove duplicate redirect test
- Move `Rating.first` to `setup (@rating = Rating.first)` or use fixtures
- Rename test to express what they are testing

```
def save(params = {})
  get :edit, :id => @rating.id
  submit_form :rating => params
end

def test_should_update_rating_on_successful_save
  save :r1=>3
  @rating.reload.r1.should == 3 #reloading from db to see if it really was stored
end

def test_should_redirect_to_index_on_successful_save
  save
  assert_redirected_to :controller => 'rating', :action => 'index'
  assert_false flash[:notice].blank?
end
```

The refactored test is lighter and more verbose. It is easier to understand and it now seems obvious, that a failing save was never tested, but before it was easy to overlook.

4.2.3 NAMING CONVENTIONS

Tests should be named for what they verify. If it is hard to find a good name, it often means too much is being tested and breaking it down in 2 cases will make finding names easier. A good scheme for naming is the assumption itself. “update should redirect me to index when successful”, this can be shortened to “update should redirect to index on success”, which results in `test_update_should_redirect_to_index_on_success`.

These names seem rather long, but they clearly state what is tested. If not, tests often end up being named `test_save`, which suggests “save is covered, no more testing required” even if the test just verifies half of the behavior. It also helps when seeing the failure output; it is instantly obvious where and why (“because it did not redirect to index”) something is broken. Other developers or the “later self” can see what was intended without reading the code, which may test something different than stated or be hard to understand.

4.3 GUIDELINES

4.3.1 LOOSE COUPLED TESTS

It is best to have a tight coupling between behavior and test, so that e.g. the model tests fails when the model breaks, which makes testing and error finding faster through fast feedback. But tests should be decoupled from ever-changing aspects like spelling, that do not influencing the behavior and where every typo means that tests have to be altered.

```
assert_equals 'Your transaction was successfull!', message #bad
assert_match / success/, message #good
```

4.3.2 FLASH MESSAGES

It is a good practice to separate flash messages into success (flash[:notice]) and failure (flash[:error]). User will faster recognize if something succeeded or not when used with different styles (e.g. green for successful actions; red for failed actions) and tests can use assert flash[:notice] to test if a success notification was sent, meaning that they will not break on wording changes, since they depend directly on the behavior.

4.3.3 DUCK TYPING

„If it walks like a duck and quacks like a duck, I would call it a duck.“ - James Whitcomb Riley

Rely on duck typing: testing something behaves in a desired way and not that it is an instance of some specific type. The methods kind_of? and respond_to? are far more robust than instance_of when changing the type of a parameter or refactoring.

4.3.4 PUSH DON'T PULL

It is best to push information and configuration into objects. They will be easier to reuse and easy to fill with mock servers/service providers/databases/... . It should be a warning sign when some Objects need to load a configuration file or pull information from the environment using something like GlobalConfig[:information]. Make all these connections with the environment as loose as possible, by pushing them in using the constructor or explicit setters (Fowler, dependency injection, 2004).

4.4 FIXTURES

VALID FIXTURES

Using invalid fixtures for something like testing edge-case behavior will increase the number of fixtures that have to be maintained and also makes maintaining all fixtures harder, because invalid fixtures cannot be distinguish from broken fixtures. Therefore it is best to only change fixtures inside the 'failure behavior' test cases. This way fewer fixtures are needed and failure cause and failure handling are kept close by. When all fixtures should be valid, it is easy to run a task to check that all fixtures are indeed valid. This saves time when introducing new model validations or model attributes. Here is an example refactoring that replaces invalid fixtures with in-place changes:

```
def test_user_should_not_be_valid_without_email
  assert_invalid users(:no_email)
end
➔ without invalid fixtures, code gets clearer and saves 1 fixture
def test_user_should_not_be_valid_without_email
  user = users(:one)
  user.email = ''
  assert_invalid user
end
```

A rake task to validate all fixtures and all models can be found at (Grosser, valid attr, 2008), its output helps to find out which fixtures are invalid and what attributes are not valid:

```
$ rake db:validate_fixtures
-- records - model --
      1x FtpAccount
     22x Movie
Movie: id=5
["Title can't be blank"]
      8x Order
     24x Rating
      1x User
```

Example output: 1 movie does not have the required title and should be corrected.

MANY FIXTURES

When large amounts of fixtures are needed, for testing speed or pagination, it is best to create them inside the actual test (def create_100_fixtures) or to use the (Preston-Werner, 2007) plugin. Fixture-scenarios allow having sets of fixture in subfolders. These specialized fixtures get combined with the normal fixtures, only for certain tests. Either way the goal is to keep the test environment free from unnecessary fixtures, which make testing slower and more complicated.

4.5 DEVELOP WITHOUT OPENING A BROWSER

Verify with tests and console first and save the browser as last method. Clicking around and filling forms is slow and leaves nothing that can be automated, while writing tests helps to become better at it and leaves an automated suite for all other developers to use. Testing first and then seeing it break in the browser, because there was no redirect/input/link/... , raises awareness for what aspects often lack testing and therefore results in better test cases and higher coverage.

4.6 TESTING VIEWS

It is cumbersome to test view markup with `assert_tag` or `assert_select` (verifying that some tags exist) and results in tests breaking when only changing view layout or structure, therefore only test view behavior not markup.

For me the things that break in views are always mistyped functions which results in errors that are visible when rendering the view, CSS problems which will only be uncovered by looking in the browser or syntax errors that are hard to see in a browser but can easily be found via `html_test` (5.3.7). Hence the only thing left to test is if the correct content is displayed and if the view logic worked.

To test the logic in the views, remove it from the views and place it in view helpers, where it according to Rails conventions, naturally belongs. Testing helpers will be easier, since most of the time they only return a line of text or a link. From my experience, the only things worth testing are that a form has all fields required for an update or create, that the page is valid HTML, that all generated links are valid and that the layout is correct.

The first 3 can be tested with `form_test_helper` (5.3.5) and `html_test` (5.3.7). When there is still logic left, test for the most relevant result (e.g. a link to go to the admin page is shown or not:

```
assert_equal 1, tags("[@href=#{admin_url}]").size
```

To illustrate the redundancy of markup validations, a simple demonstration using RSpec and 2 Users:

```
it "sould have one row for each user" do
  render '/users/index'
  response.should have_tag('table tr', :count => 2)
end
```

Is wrong and unnecessary:

- because the table has 3 rows (2 Users + Heading)
- `tr` is always inside a table, or it would raise a syntax error (`html_test` (5.3.7)).

It is hard to maintain, since it was not asserted that 2 users where rendered, but that 2 rows exists, which could contain anything and nothing.

- When adding another heading, the test has to be changed
- When adding a table to describe the contents, a new selector has to be found
- When refactoring using divs, everything breaks even though the same information is displayed
- When the resulting table cells are empty, nothing fails

4.7 CONCLUSION

When not using the right techniques testing efforts may be fruitless. When done right, testing will not only lead to verified code, but to better, loosely coupled code. Focus on the risky parts of the application and do not spend time testing each and every part, especially when some parts are ever-changing (e.g. HTML structure) without effect on the behavior. Since time is limited one should opt for a maximum amount of security per written test case. On an imaginary list of risk factors where $\text{risk} = (\text{amount of money lost}) * (\text{chance this will happen})$ anything below a certain threshold should be discarded.

5. TESTING TOOLS

5.1 INTRODUCTION

Rails comes with a rich assortment of testing tools and a well-defined test layout structure. For small application that often is sufficient. When an application gets larger, tests usually grow faster than the application itself, which often results in an unorganized test structure and duplication. Problems arise that do not seem to be testable (e.g. caching, emails, Javascript...) and especially when more than a single developer works on the codebase a better organization repays fast. The following chapter shows many possibilities to write dryer and more efficient tests, how to structure tests, how to see which parts lack testing, which possibilities exist for Javascript testing and an alternative testing framework.

5.2 INSTALLATION TIPS

5.2.1 PLUGINS

It is recommended to document all plugins and gems installed, to remember how the system was set up, repeat a setup or help coworkers understand the dependencies.

There are 3 often used ways to install a plugin, 2 of them are not recommended since they will destabilize the environment. Here they are, sorted from bad to good.

AUTOMATICALLY UPDATING DEPENDENCIES

With `script/plugin -x` or `svn:external` dependencies will always be up to date. It sounds good, to always have the latest version of a plugin, but when the author decides that a 'bit of testing' is enough for the feature or the API changes, one day before the plugin is installed to the server, there will be no clue why suddenly the application fails.

Furthermore, changes in the plugin cannot be saved or committed. Every newly checked out version will have to be changed again. More on the subject of safely changing and modifying plugins can be found at (Err, evil-twin, 2007)

USING PLAIN SCRIPT/INSTALL

This will make a local copy of the plugin and is by far the most often used method, but also has drawbacks. In case the plugin was modified locally, an upgrade to the newer version with a new 'script/plugin install' will remove all SVN information, meaning the plugin has to be completely removed, committed and re-added, then all changes have to be reapplied, which can fail if the changes were not properly documented and it often means a lot of work.

PISTON

A combination of all the advantages and none of the disadvantages can be found in piston. The plugin is copied into the local application, can be committed and can be kept up to date with the latest plugin revision. Running 'piston update' works like a normal SVN update and warns about conflicts and merges inside the plugin resulting from local changes that were made. Only piston will treat the plugin as an external repository, the local version control treats it as just another folder.

```
gem install piston
svn update #before each import, to stop piston from complaining

piston import svn://somehost.com/svn/plugins/my_plugin \
  vendor/plugins/my_plugin
piston update vendor/plugins/my_plugin #get new plugin version
```

- 'piston status' to see which repository's have changed and which plugins are locally modified
- 'piston convert /folder/' convert an existing svn:externals folder into a piston-managed folder
- 'piston lock /folder/' lock one of the piston-managed folders, so that all others can be updated, but the one plugin whose new version does not work, stays the same

Piston also works with GIT version controlled project and plugins. More commands and help can be found at the projects homepage (Beausoleil, 2008).

5.2.2 GEMS

To clearly document which gems an application needs and to always check if they are installed, `config.gem "gem_name"` is be added to `config/environment.rb`. To install missing gems use `rake gems:install`, or use `rake gems:unpack` to generate a local copy that does not depend on the global system configuration. If the name of the gem does not equal the name of the required library, add the libraries name too (see example below). Documenting the version of a gem is important since gems are often updated by accident when another installed gem requires a newer version of a gem that is already present. The gems version must be added or code may break on one server and run fine on another.

```
config.gem "SQS", :version => "0.1.7", :lib => 'sqs'.
```


5.3 TOOLS

5.3.1 CACHE_TEST

Testing caching-logic, like sweepers or other cache-expiry methods, is rather hard since the test environment has turned caching off and therefore will never cache a page and cannot test if it would have been cached or expired. With the (Merenne, 2006) plugin these tests become possible, to install it copy the extracted archive into vendor/plugins.

```
def test_expiring
  assert_expire_pages("ratings ", "ratings/1") do |*urls|
    delete "ratings/1"
  end
end
```

This test makes sure that the index and the show actions are expired when the corresponding model gets deleted.

5.3.2 RED GREEN

The normal test results are hard to read, since they are only displayed in black-on-white, which makes seeing the difference between a failed, pending or passed test run difficult. Therefore it is recommended to use RedGreen, a very simple plugin that changes test results to red (failed) / yellow (pending) / green (passed), so the test result is instantly visible without further reading the output.

```
sudo gem install RedGreen
rg test/unit/rating_test.rb
....
Finished in 2.358123 seconds.
4 tests, 13 assertions, 0 failures, 0 error
```

To see this colorful output, test must be run with the rg command or require 'redgreen' is added to test/test_helper.rb. (Sanheim, Red Green, 2007)

5.3.3 BUG TO TESTCASE (LAZINESS)

Laziness is helpful way to convert most bug-reports into test cases, since every failing request will print a small test case to repeat it, on the normal Rails error page. This test case can simply be copied and run to see if the error is reproducible. Most of the time it is only a starting point, but is very helpful if a lot of parameters are passed to a request or it is not sure which of the parameters (session/cookies/headers/...) has caused the error.

```
script/plugin install http://svn.extendviget.com/lab/laziness/trunk
```

An example output from one of my error pages that was automatically created from laziness and could reproduce the error:

```
def test_get_rating_edit_should_not_raise_activerecord_recordnotfound_exception
  assert_nothing_raised(ActiveRecord::RecordNotFound) do
    get :edit, {"id"=>"1"}, {:user_id=>nil, :return_to=>"/orders/1"}, {},
    {"_session_id"=>["..."]}
  end
end
```

(Scofield, 2008)

5.3.4 ZENTEST

ZenTest is a gem that provides a whole collection of useful tools and test helpers, the best parts are Autotest and Test::Rails, which will be cover here. Additionally it comes with a tool called multi-ruby that can easily test code in multiple versions of ruby, very useful when building libraries, plugins or gems.

```
sudo gem install ZenTest
```

AUTOTEST

When working with continuous integration there is nothing better than continuous testing. With Autotest this is easy, since it will run the matching test suite for every file that is changed. When user.rb is saved, it runs units/user_test.rb, the only thing developers have to do is to watch their console to see if tests ran successfully.

```
Editing movie_controller.rb

Running test/movies_controller_test.rb...
Finished in 15.106864 seconds.
24 tests, 49 assertions, 0 failures, 0 errors
```

Once a test fails Autotest will re-run only this failed one, not the whole suite, so the problem can be narrowed down while having fast feedback. It also strips most of the redundant error output (framework-trace) and only leaves the applications own backtrace. After the last failure has been removed, autotest runs the complete suite again, to see if any new errors have been introduced.

It is even possible to stop looking at the console and make Autotest play a sound when failing (Foz works, 2007) or raise a desktop notification (Grosser, Autotest notifications, 2008).

TEST::RAILS

Test::Rails introduces new testing possibilities, helpers, assertions and Test classes. More in-depth knowledge can be found at (ZenTest, 2006). It supports the idea of separating functional tests (Rails integrated views and controller tests) into separate controller and view tests, which follows the basic principles of unit testing, to test as small as possible. When modifying views, controller tests should not fail and when controllers change, view-tests should not fail. Testing the interaction of views and controllers, like submitting forms or following a redirect, is left for the integration tests.

Test::Rails::ControllerTestCase

The ControllerTestCase is responsible for assuring that all values get assigned (assert_assigned), that the right model actions were called (save/create...) and the flash/session is set correctly. It is an ideal place to use mocking (see Mocha 5.3.8) since not all attributes of an object need to be known (they are never displayed). Additionally controllers can be separated from all validation logic (failing actions are tested by mocking `model.valid?` to false).

Steps for migrating from a normal functional test to ControllerTestCase:

- Replace old extends with ControllerTestCase
- Remove @response, @request, @controller lines from setup and call super
- When the test does not follow the naming of the controller tested, set the controller name by hand, e.g. with this snippet, which infers the current controller from the file the test lies in:

```
@controller_class_name = File.basename(__FILE__, "_test.rb").classify
```

More info on ControllerTestCase setup and usage can be found at (Eric, 2006)

Test::Rails::ViewTestCase

This type of TestCase sits between a controller and its view. It provides the parameters for the view and tests only what the view does with the given input. This way edge-case behavior of views can be tested without building the normally needed support-code like controller requests. Partial which are used by different views can be tested with varying input and the application layout can be inspected on its own by rendering it with an empty content: `render :text => '', :layout => 'application'`

ViewTestCase comes with many specialized assertions like `assert_links_to`, `assert_post_form`, `assert_input...`. This example Testcase shows how a test is set up, by assigning the variables the controller normally would provide and then testing if the form is posted to `/movies` when the `'Delete!'` button is pressed.

```
class MovieViewTest < Test::Rails::ViewTestCase
  def test_show_has_working_delete_form
    assigns[:loggedin_user] = users(:herbert)
    assigns[:movie] = movies(:two)
    render 'show'
    assert_submit '/movies', 'Delete!'
  end
end
```

This approach is time-consuming when only a simple action and its resulting view should be tested but the more complex controllers get and the more edge cases views have to handle, the more appealing this approach becomes. It is possible to use a mixture of functional and pure controller/view tests, but once started, it is wise to switch all functional to controller/view tests, since then `rails_test_audit` can be used.

rails_test_audit

When used in combination ViewTestCase and ControllerTestCase supply a new feature: running `rake rails_test_audit` will show which variables have been tested for in the controller (by `assert_assigned`), that have not been supplied in the ViewTestCase (by `assign`) and vice versa, this helps to find unnecessarily assigned variables or missing input for views.

5.3.5 FORM_TEST_HELPER

Form_test_helper can submit a form that was created in a previous request and test if it transmits all necessary fields to create or update a model. It also reduces the work for testing a post request, by using the values already filled in the form and thereby validating that the form has all necessary fields.

```
script/plugin install http://form-test-helper.googlecode.com/svn/form_test_helper/
```

A small example that fills a form and then submits it, more can be found at (Garber, 2008).

```
submit_form do |form|
  form.movie.title = 'Test movie'
  form.movie.plublic.uncheck #checkbox handling
end
#OR
movie = {'title'=>'Test movie', 'public' =>false}
submit_form {|form| form.movie.update(movie)}

assert_response :success
```

With the help of syntactic sugar it is even possible to click a link from inside tests, making it possible to test real user interaction and not only calling urls that possibly never were displayed, thereby also testing that the link exists on the current page.

```
#test/test_helper.rb
def click(text)
  select_link(text).click
end
```

On revision 69 it is necessary to apply this patch for click to work with Rails 2.1 new post/put/delete links.

```
#vendor/plugins/form_test_helper/lib/form_test_helper.rb
-if self["onclick"] && self["onclick"] =~ /'_method'.*'value', '(\w+)'/
-  $1.to_sym
+if self["onclick"] && self["onclick"] =~ /\.\method = '(.*?)'/
+  $1.downcase.to_sym
```

5.3.6 RAILSTIDY

Railstidy is a plugin that can validate existing html pages on a server or validate all the pages that are generated during test runs for compliance with the WC3 standards. For more details see (Merenne, Rails tidy, 2006). Installation is rather complicated, here are the steps to get it working on Ubuntu 8.10.

```
#installation from a Rails directory
sudo apt-get install tidy

sudo gem install tidy
wget http://www.cosinux.org/~dam/projects/rails-tidy/rails_tidy-0.2/tidy.patch
sudo patch /var/lib/gems/1.8/gems/tidy-1.1.2/lib/tidy/tidybuf.rb < tidy.patch

cd vendor/plugins/
wget http://www.cosinux.org/~dam/projects/rails-tidy/rails_tidy-0.2.tar.bz2
tar -xf rails_tidy-0.2.tar.bz2
rm rails_tidy-0.2.tar.bz2
```

To try it out, validate all current view templates:

```
rake test:templates
/home/data/projekte/short/app/views/movie/list.rhtml      ERRORS
/home/data/projekte/short/app/views/movie/edit.rhtml      OK
```

These results show where problems lie, but to narrow down the search each individual file has to be parsed:

```
rake test:templates FILE=app/views/movie/list.rhtml
app/views/movie/list.rhtml                                ERRORS
line 6 column 1 - warning: <br> element not empty or not closed
line 10 column 1 - warning: <table> lacks "summary" attribute
...
2 warnings, 0 errors were found!
```

After all problems are resolved, RailsTidy can easily protect the project from inserting any new errors by always validating the HTML output generated during normal test runs, by adding a global after-test callback that will validate the rendered page.

```
#test_helper.rb
class Test::Unit::TestCase
  def teardown
    assert_tidy if @response
  end
end

Tidy detected html errors in response body:
-----HTML of whole page-----
line 65 column 33 - warning: inserting implicit <p>
line 65 column 33 - warning: trimming empty <p>
```

Some of the errors result from using html helpers like `form_tag`, so this output sometimes indicates an error when there was nothing done wrong by the user, which can be frustrating. To silence unwanted warnings like “table lacks “summary” attribute”, it is possible to configure RailsTidy with custom ignore patterns, by modifying `config/tidy.rc`. A list of possible configuration options can be found at (Railstidy, 2008).

5.3.7 HTML_TEST

Contrary to RailsTidy, `html_test` is not a real validation library, but a tool to make validation testing simple and pain-free. It can use up to 3 different validators, validate the response of every test-request, suppress unwanted warnings and check all rendered links to see if they would direct to an existing resource (no 404). To make use of the HTML validation, first RailsTidy or any other of the 3 supported validators has to be installed and then `html_test`:

```
script/plugin install http://htmltest.googlecode.com/svn/trunk/html_test
```

This configuration will validate all responses with RailsTidy, ignore many unhelpful or outdated warnings and validate generated links as well as redirects:

```
#insert into test/test_helper.rb
#validate every request
ApplicationController.validate_all = true
ApplicationController.validators = [:tidy]

#ignore common warnings
Html::Test::Validator.tidy_ignore_list = [
  /<table> lacks "summary" attribute/,
  /trimming empty <fieldset>/, #errors_on missing -> empty fieldset
  /line 1.*warning: inserting missing 'title' /, #redirect html has no title....
  /Warning: replacing invalid character code 130/, #€ not supported...
]

#check urls
ApplicationController.check_urls = true
ApplicationController.check_redirects = true
```

It is also possible to check URLs on a production server by letting `html_test` check all links on a page. This does not allow for detailed configuration (e.g. which warnings to ignore), so the output will contain many unwanted warnings, nevertheless it is a good tool to find links to non-existent pages.

```
vendor/plugins/html_test/script/validate http://my.blog.com --validators tidy
```

5.3.8 MOCHA

Mocha is a Mocking / Stubbing framework, which can help to separate controller and view tests from database and validations. With Mocha all model specific aspects, like validations and environment requirements, can be separate from unit tests. Therefore the controller tests can validate pure request/response logic, without worrying that e.g. a User needs an Email to be saved. All complex or time-consuming model and database operations can be skipped.

```
script/plugin install svn://rubyforge.org/var/svn/mocha/trunk
```

A basic test to verify that the correct model was assigned by the controller:

```
def test_edit_should_assign_rating
  rating = Rating.new
  Rating.expects(:find).with('1').returns(rating)

  get :edit, :id => '1'
  assert_equals rating, assigns[:rating]
end
```

If all test cases use Mocha, stubbing can be placed inside the setup:

```
def setup
  @rating = Rating.new
  Rating.stubs(:find).returns(@rating)
end

def test_edit_should_assign_rating
  get :edit, :id => 1
  assert_equals @rating, assigns[:rating]
end
```

The difference is that the first test will fail if find is called with 2 or find is not called or find is called twice, since it is expected that find is called exactly once with '1' nothing else. Whereas the second test only verifies that the rating is assigned, not caring from where it came.

Mocking a method call like find with a specific parameter like stubs(:find).with(:all) will make any other call to this method raise a failure, therefore all calls to find that will be used inside the test must be mocked as well (find :first, find 13...). More details can be found at the projects homepage (Mead, 2008).

5.3.9 RCOV

RCov is a code coverage analyzer for ruby, which can be integrated with Rails through the RCov Rails plugin. The RCov plugin can be used to analyze code coverage of application tests, without learning all RCov command line options. The result is a HTML file for every ruby file under test, with detailed execution coverage, and an index.html including the overall coverage for all tested files.

```
wget http://rubyforge.org/frs/download.php/28270/rcov-0.8.1.2.tar.gz
tar -xf rcov-0.8.1.2.tar.gz
sudo ruby rcov-0.8.1.2/setup.rb
rm -rf rcov-0.8.1.2
```

```
script/plugin install http://svn.codahale.com/rails_rcov
```

To run the whole test suite and generate a coverage report afterwards, execute “rake test:rcov”. The most useful parameters for this command are:

- Show only selected parts: SHOW_ONLY=m,l,c,v (model,lib,controller,view)
- Run a single test: rcov FILE --rails (the --rails option will ignore config and environment)
- Add RCov parameters: RCOV_PARAMS=""
- Sort by coverage: COV_PARAMS="--sort=coverage"
- Hide fully covered: RCOV_PARAMS="--only-uncovered"

Ignoring all standard libraries helps keeping the output to a manageable size. When using the command line, this can be achieved by adding RCOV_PARAMS="--exclude '/var|/usr/'", where /var/ and /usr/ are the folders that contain libraries and gems. When using rake, this can be done the same way, but is a bit repetitive. For this purpose it can be simpler to directly modify the plugin.

```
#vendor/plugins/rails_rcov/tasks/rails_rcov.rake @ line ~60
  if show_only.any?
    reg_exp = ['/var/', '/usr/'] #CHANGED
    for show_type in show_only
```

Additional usage information and setup instructions can be found at the author’s homepage (Hale, 2006).

5.4 RSpec

5.4.1 RSpec ON RAILS

```
sudo gem install rspec
sudo gem install rspec-rails
script/generate rspec
```

RSpec focuses on BDD (Behavior Driven Design). When TDD is done right, it is essentially the same as BDD. The problem RSpec is trying to solve lies mostly in syntax and task splitting. Normal test verify that something is the way it should be, whereas the idea behind of TDD is that it is first stated what should work and then it is built. Therefore RSpec uses a syntax that focuses on 'should' and not assert.

A main part of RSpec is the story framework, which makes it possible to let customers write and review tests, by using normal English sentences as steps. Once a step is programmed, it can be combined freely in any number of stories.

RSpec uses different words for test related things. A test is called spec, a test case is called example and integration tests are called stories story. Stories are most often used for integration testing, but they also could be used for other test areas. The best part about Rspec is that its syntax almost reads like English, which improves readability and makes it easy to write.

A comparison of Test::Unit and RSpec syntax for the same test:

```
assert_equal 'new', rating.status #Test::Unit
@rating.status.should == 'new'    #RSpec
```

STORIES

The most simple and stunning part of RSpec is a story, it is readable even to non-programmers and can be build from normal sentences It's structure was derived from story based XP development and it can consist of many scenarios that illustrate a single story in detail. The scenarios consist of any combination of 'When', 'Then' and 'Given' sentences, where an 'And' sentences is translated to the last used identifier, meaning that Given x, And y is the same as Given x, Given y.

```
Story: transfer from savings to checking account
  As a savings account holder
  I want to transfer money from my savings account to my checking account
  So that I can get cash easily from an ATM

  Scenario: savings account has sufficient funds
    Given my savings account balance is $100
    And my checking account balance is $10
    when I transfer $20 from savings to checking
    Then my savings account balance should be $80
    And my checking account balance should be $30
```

To transform this story into a working test case, it must be defined what 'Given my savings account balance is \$100' means in program terms, this is done in the steps file that belongs to any story. This way an executive or designer can write stories and later the programmer can fill them with life.

Steps used to run this example story:

```
steps_for(:accounts) do
  Given("my $account_type account balance is $amount") do \
    |account_type, amount|
      create_account(account_type, amount)
    end
  When("I transfer $amount from $source_account to $target_account") do \
    |amount, source_account, target_account|
      get_account(source_account).transfer(amount).to(get_account(target_account))
    end
  Then("my $account_type account balance should be $amount") do \
    |account_type, amount|
      get_account(account_type).should have_a_balance_of(amount)
    end
end
```

(RSpec, 2008)

When steps for all sentences used are defined, any number of test cases can be run with them. Not all steps have to be built at the same time, missing steps will be marked as pending and the story can be executed until the first missing step is reached. Users that like the story feature, but want to stay with the Test::Unit syntax, can use normal 'assert' syntax in step definitions.

MODELS, CONTROLLERS, VIEWS, HELPERS

Stories are not everything; they are just the top part, the former integration tests. For all other 'small' parts normal RSpec tests can be used. These are split into 4 different categories: Model, Controller, View and Helpers. Controllers and views are the former parts of a standard Rails functional test, like `Test::Rails` (5.3.4), controller tests only verify that i.e. an object was assigned and a view was rendered, whereas a view test gets predefined assigns and verifies that it renders correctly. Only when used with `integrate_views`, controller tests behave like functional (`Test::Unit`) tests.

Since RSpec tests can work like normal unit/functional tests they are a good point to start learning RSpec syntax. A helpful learning recourse is the RSpec cheat sheet (Astels, 2006) and the RSpec converting table (Meester, 2008).

A basic controller spec, not very different from `Test::Unit`:

```
describe RatingController do
  integrate_views #render views wen executing a controller action
  fixtures :ratings

  before do
    @rating = ratings(:one)
  end

  it 'should render index' do
    get :index
    response.should render_template 'index'
  end
end
```

(Requiring the controller is no longer necessary and the 3 old `@controller`, `@request`, `@response` statements can be left out too.)

STUB AND MOCK

RSpec contains a mocking framework whose syntax is not as clean as Mochas (5.3.8), but it offers comparable abilities. A comparison of RSpec and Mocha syntax:

```
Rating.stub!(:new).and_return(@rating) #RSpec
Rating.stubs(:new).returns(@rating)    #Mocha
```

To stay with Mocha mocking, this configuration must be added to `spec/spec_helper.rb`.

```
Spec::Runner.configure do |config|
  config.mock_with :mocha
end
```

RUNNING RSpec

Rspec tests can be run via rake “rake spec” and only a subset of the tests can be run with e.g. “rake spec:models”, the supported subsets are: models, controllers, views, helpers, plugins and app, all, where “app” means everything but “plugins”. Additionally single tests can be run with “spec path_to_test_file”. Test execution can be slow, since the whole Rails framework has to be loaded before tests are run. For faster test execution it is possible to host an RSpec server. While this server is running, testing no longer requires starting up the Rails framework and therefore is faster, especially for small tests.

```
script/spec_server -d      #run server as deamon (kill: 'ps -A', 'kill ID')
echo '--drb' >> spec/spec.opts #add -drb to spec options
rake spec                  #run spec as normal
```

RUNNING STORIES

Getting started with stories is rather hard, since there is no generator to help. Thereupon the default helper.rb and all.rb are not suited for a large set of stories, because they force all stories and steps to share a common namespace.

An example helper.rb that supports placing all stories in stories/stories/{name}.txt and all steps in stories/steps/{name}.rb, so that they are better organized and can be found easily:

```
ENV["RAILS_ENV"] = "test"
require File.expand_path(File.dirname(__FILE__) + "../config/environment")
require 'spec/rails/story_adapter'

#load all steps
Dir[File.join(File.dirname(__FILE__), "steps", "*.rb")].each do |file|
  require file
end

#run a story
def run_story(file_name)
  run File.join(File.dirname(__FILE__), "stories", "#{file_name}.txt"), :type =>
  RailsStory
end
```

The script that executes the story lies in stories/{name}.rb and is executed with ‘ruby stories/{name}.rb’. For rating stories the general steps and the specialized rating steps are needed, so running these stories would look like this:

```
require File.join(File.dirname(__FILE__), "helper")
with_steps_for :rating, :general do
  run_story 'rating'
end
```

5.4.2 RSPEC CONVERTER

Converting old Test::Unit test cases to RSpec by hand can be very helpful in learning RSpec syntax, but once the syntax is understood, it is a repetitive task. Therefore the Test::Unit – RSpec converter can be used to automate this process. It can be installed via

```
script/plugin install \  
http://svn.davidjrice.co.uk/svn/projects/plugins/test_unit_to_rspec_converter
```

and is executed with “rake convert_to_rspec”.

With the converter, all tests will be converted to RSpec, thereby overwriting all files that happen to be where the new tests should be created. Not all old assertions need to be replaced, since RSpec can work with Test::Unit ‘assert’. But before everything runs a bit of hand work is needed:

1. Copy old code from test/test_helper.rb to spec/spec_helper.rb (leave include’s outside of Spec::Runner.configure do)
2. Change spec/spec_helper.rb ‘config.fixture_path =’ to test/fixtures OR move all fixtures from test to spec (svn mv test/fixtures spec/fixtures)
3. Correct syntax errors in converted files

When all tests run successfully again, test code should be refactored to make use of the superior RSpec organization mechanisms like e.g. “describe” and “should raise_error”. More information on setup and usage can be found on the author’s blog at (Rice, 2007).

5.4.3 ALTERNATIVES TO COMPLETELY SWITCHING

For those that like RSpec syntax, but can or will not make a complete switch, there are 2 alternatives. The first is test_spec, an RSpec like syntax replacement for Test::Unit. It allows mixing of RSpec coding style and normal tests. It is setup by installing the plugin and requiring the test_spec helpers:

```
sudo gem install test-spec  
script/plugins install \  
http://svn.techno-weenie.net/projects/plugins/test_spec_on_rails/  
  
#test/test_helper.rb  
require 'test/spec/rails'
```

Syntax cheat sheets can be found at (Err, 2007). A simple syntax example that shows some of the more unique abilities and helpers of test_spec:

```
@rating.should.validate      #assert @rating.valid?  
response.should.be.redirected #assert_response :redirect
```

Additional instructions can be found on the test_spec homepage at (Sanheim, BDD in Rails, 2006).

The second alternative to a full RSpec conversion is simply running Test::Unit files with RSpec. When Test::Unit files are run as a RSpec tests, they behave exactly like before, but can use e.g. should / should_not for assertion. So is is possible to drop all functional tests into spec/controllers, copy the fixtures, replace all test/xxx with spec/xxx and tests should pass as if nothing has happened.

5.5 JAVASCRIPT

5.5.1 UNOBTUSIVE JS

Javascript can be separated into 2 categories, obtrusive and unobtrusive. Here is a small example to show the difference.

```
<div onclick="hello()"> text </div>
```

This is obtrusive, since it changes the structure of the HTML document (the Document Object Model, further referred to as DOM) by inserting attributes (onclick,onmouseover,...) to the DOM that do not describe its structure. Unobtrusive on the other hand does not modify the HTML structure:

```
<head>
  <script type="text/javascript" src="jquery.js"></script>
  <script>
    jQuery(function($) {
      $('#test').click(function() {
        $(this).html('I was clicked');
      });
      return false;
    })
  </script>
</head>
<body>
  <a id="test" href="#">click me</a>
</body>
```

Figure 3: Example of unobtrusive Javascript

- 1 load jQuery library
- 2 "when the page is loaded, start this function and pass the jQuery object as parameter \$ to it
- 3 when test is clicked, change its 'innerHTML' to 'I was clicked'

Using this setup (step 1 + 2) can be overhead for overly simple cases like this. But for larger cases or projects, not using it leads to ever-increasing complexity. In general there should be 3 Layers in the Frontend: HTML(Structure), CSS(Style), Javascript(Behavior). Unobtrusive in this context means: "separate Javascript from HTML like CSS".

Building obtrusive has many disadvantages, some of the most obvious are that callbacks like "onClick" can only call a single method and not multiple, that Javascript code is repeated several times which makes maintenance harder, that files normally get larger through the repetition, that Javascript cannot simply be turned off since it is embedded everywhere and it is hard to move behavior from one component to another to reuse it.

With unobtrusive Javascript behavior is cleanly separated from HTML, but it is not possible to test that it works or even if it is there after all from inside Rails, by writing something like "test that HTML changes when link #test is clicked" since Rails only knows the pure HTML output that came from the controller, and has no idea of any attached Javascript behavior.

Simple structures written in obtrusive Javascript are testable by "assert there is an onclick attribute that looks like /\$('#test').click(/". But this kind of testing does not help, since it cannot prove the script will work, just be sure that it looks like it could work. What nevertheless some frameworks do e.g. (Cauldwell, 2007).

5.5.2 JAVASCRIPT_TEST

```
script/plugin install http://dev.rubyonrails.org/svn/rails/plugins/javascript_test/
```

With javascript_test it is possible to run Javascript tests automatically in up to 4 different browsers. To use this feature all supported browsers should be installed (currently supported: IE and Firefox for Windows, Firefox and Safari for Mac, Firefox and Konqueror for Linux). On Windows all browsers must be installed in their default location and tests in Internet Explorer have to be closed by hand. It is hard to get javascript_test running properly, but once it works a lot of repetition can be saved.

The plugin also allows to generate a Javascript test scaffold, with:

```
script/generate javascript_test my_app
create test/javascript/my_app_test.html ...
```

This gives a good starting point for test setup and organization, even if tests are run by hand (loading them in the browser) and do not get automated.

5.5.3 UNITTEST.JS

Unittest.js is a test framework build using prototype.js, it is easy to get started and has graphic output. An example test setup would look like this:

```
...
<script src="path/to/prototype.js"></script>
<script src="path/to/unittest.js"></script>
...
<div id="testlog"> </div>
<div>Test text</div>
<script>
  new Test.Unit.Runner({
    setUp: function() {},
    testTruth: function () {
      this.assertEqual(2, 1+1);
    });
  });
</script>
```

When executed by opening the file in a browser, the test results are displayed:

script.aculo.us Unit test file

Test dynamic loading

1 tests, 1 assertions, 0 failures, 0 errors

Status	Test	Message
passed	testTruth	1 assertions, 0 failures, 0 errors

Figure 4 Result of JUnit test run

More information can be found at (script.aculo.us, 2006) and a live demo can be found at (Sunnyfortuna, 2006). A simple way to get started with Unittest.js is to save the demo page locally, which includes all Javascript and CSS needed, and then write some example tests with this working setup.

5.5.4 JQUNIT

JQUnit is a Javascript testing framework that was originally developed by the jQuery team, and was given xUnit and TestCase support as a better organization (e.g. no global namespace pollution) by Colin Clark and me. Its current home is <http://code.google.com/p/jqunit/> and we hope that it will be integrated into jQuery core soon.

JQUnit is not a xUnit framework, for example `assertEquals(1,output())` is written as `equals(output(),1)`. It still supports all standard xUnit commands, so it should be easy to adopt. The framework uses the full power of Javascript by breaking out of xUnit conventions:

- The whole page will be reset after each test, so there almost never is a setup/teardown needed
- Click an test output line to unfold Test-Results (see each assertion)
- Double Click a test to rerun it
- Open `test.html?xxx` will run only tests that match `/xxx/`

It is possible to use TestCase or to just start writing tests with `test()`.

```
with(jqUnit){
  module('with local interface');
  test('test 1', function(){
    ok(true);
  });
};

//same Using TestCase
var t = new jqUnit.TestCase('TestCase',function(){
  /*setup*/
},function(){
  /*teardown*/
});
t.test('test 1',function(){with(jqUnit){ this.ok(true) }});
```

Running the test by opening it in a browser shows this result:

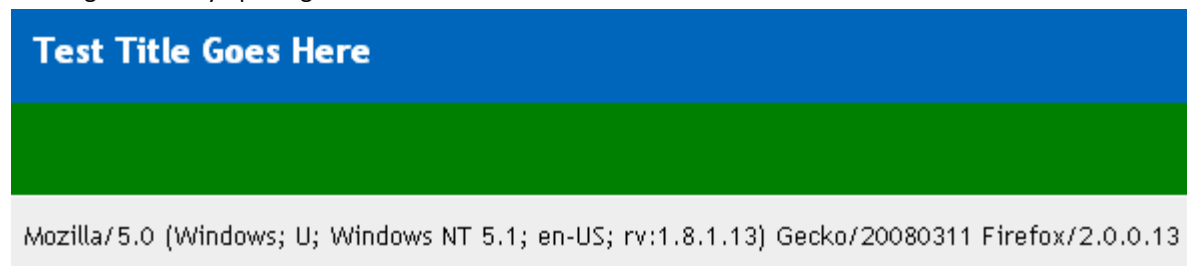


Figure 5: Result of JQUnit test run

The advantage of using TestCase is that setup and teardown will run each time a test is run and this way tests can be separated into different contexts. To get started, download the demo and source packaged from the JQUnit homepage at (Grosser, JQUnit, 2008).

5.5.5 SELENIUM

```
script/plugin install http://svn.openqa.org/svn/selenium-on-rails/selenium-on-rails
```

For detailed installation instructions and 'getting started' help can be found at (Selenium, 2008).

Selenium is a browser-automation tool. It comes with a Rails plugin and an IDE to create Unittests simply by browsing a website and hitting the record button. Recorded browsing sessions can then be automated inside a normal Unittest using Seleniums 'Remote Control' Browser API. All that is left to the developer is inserting the right assertions for the displayed content and for the Data that is modified.

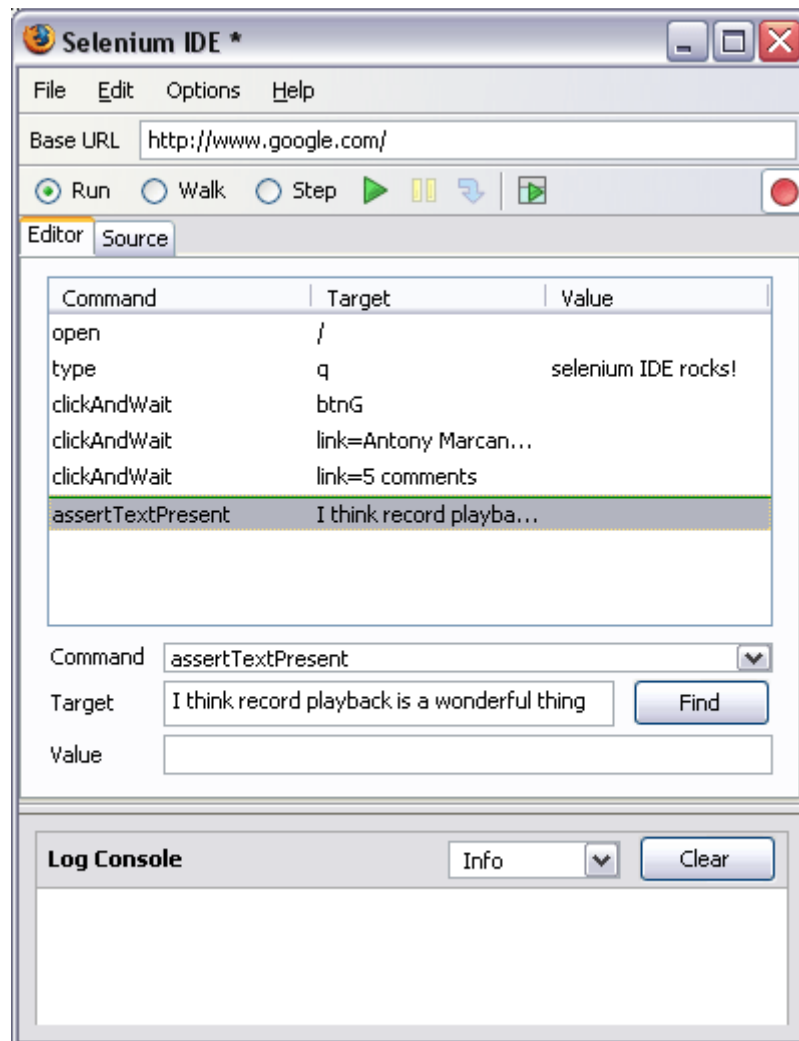


Figure 6: Selenium IDE inside Firefox, actions are being recorded and can be enhanced through assertions directly in the browser

Selenium Remote Control Server can be used to test a website automatically. It runs on any operating system and executes test calls from test clients. This way testing of any browser on any system is possible since a single test client can use multiple Remote Control Servers.

A few tips to get started:

- Pressing 'return' is "key_press locator, '/13' "
- Locators can be ids(#search = 'search') or CSS (input with name search="//input[@name='search']")

To evaluate if Selenium is the right tool, start with the standalone Ruby Client that comes with the Remote Control Server. Start the server with "java -jar selenium-server.jar" and then execute tests.

An example Selenium testcase, that connects to a local Remote Server on port 4444 to test if the ebay search page has the text "My Ebay" in the div with the id "MyEbay":

```
require 'test/unit';require 'selenium'
class ExampleTest < Test::Unit::TestCase
  include SeleniumHelper

  def setup
    ie = '*iexplore C:\Programme\Internet Explorer\IEXPLORE.EXE'
    url = "http://www.ebay.com/"

    @selenium = Selenium::SeleniumDriver.new("localhost", 4444, ie, url, 6000);
    @selenium.start
  end

  def teardown
    @selenium.stop
  end

  def test_small
    open "/"
    type 'satitle', 'all'
    click '//input[@value=\'search\']'
    sleep 7
    assert_equal('My Ebay', get_text('MyEbay'))
  end
end
```

In conclusion, Selenium is a powerful tool, that is hard to setup and maintain, but it allows simultaneous browser testing on many remote servers. It is the only tool so far that delivers this ease of TestCase construction through the integrated Selenium IDE and clients in 7 programming languages.

5.6 HELPERS AND TASKS

5.6.1 VALID_ATTRIBUTES

Validation tests are repetitive tasks and can be simplified with (Grosser, valid attributes plugin, 2009)

```
script/plugin install git://github.com/grosser/valid_attributes.git
```

With a very simple syntax, the most common validation rules can be tested. Here for example is tested that the login is invalid when empty or nil and email is invalid when empty, nil or it is not email address:

```
assert_invalid_attributes User, :login => ['', nil], :email => [nil, '', 'ohno']
```

Each failing test results in a readable failure message like `<User.email> expected to be invalid when set to <ohno>` which shows which field failed with which test data and helps to easily pinpoint the problem.

Additionally it can be used to generate customized objects for other testing scenarios:

```
user = valid User | user = valid(User, :email=>'sam@web.de')
```

5.6.2 AGILEDox

AgileDox converts test cases into documentation and inserts this documentation into the corresponding models/controllers, so that developers always have a reference when browsing source code.

It is installed by copying the `agiledox.rake` file into `lib/tasks` from (Grosser, AgileDox, 2008). Thereafter it is possible to run `rake dox` or `rake spec:models:dox` or `rake test:units:dox` to get tests displayed in this format:

```
A User:
- should not be valid without login
- should not be valid without email
A Users Controller's:
'new' action:
- should succeed
'edit' action:
- should succeed
A /users/edit:
- should show errors
```

Changing the options `:write` value in `agiledox.rake` to `true` will also write this output into the comment block of the tested classes, thereby preserving any preexisting comments. Only the areas between `#AGILEDox` will be updated when new tests are documented. The generated comment block looks like this:

```
#AGILEDox
#A User:
# - should not be valid without a name
# - should be valid without an email
#AGILEDox
class User < ActiveRecord::Base
...
end
```

5.6.3 SINGLE TEST

```
script/plugin install git://github.com/grosser/single_test.git
```

Running a whole test suite can be time-consuming when the suite is big or slow. To only run a single test file, one has to know the command-line interface of the testing framework and type the path to the test file. Single test makes it possible to run any test with a simple rake-like syntax and thereby keeps repetitive test runs fast:

```
rake test:rating:should_rate_  
→ run every test that matches /should_rate_/ in test/units/rating_test.rb  
rake test:ratings_c  
→ run test/functionals/ratings_controller_test.rb
```

It also works nicely with RSpec, where the syntax translates to spec:.*.

5.7 PLUGIN RECOURCES

Many of the plugins and tools have alternatives that do similar things, and all the time new plugins and gems are created. To know which plugins exist and which where newly created and maybe solve a problem that has been ignored since a long time can save many hours of work and bring the security of a proven technology. To stay current on plugins news, subscribing to some of these plugin resources is mandatory.

<http://wiki.rubyonrails.org/rails/pages/Testing+Plugins>

<http://svn.techno-weenie.net/projects/plugins/>

<http://dev.rubyonrails.org/svn/rails/plugins/>

<http://wiki.rubyonrails.com/rails/pages/plugins>

<http://topfunky.net/svn/plugins>

<http://www.agilewebdevelopment.com/plugins/index>

5.8 CONCLUSION

With these tools testing can be done more efficient, covering more application logic with fewer, robust tests. Readability and organization also increase when tools are used rather than endless lines of setup code that only distract the developer. These are of course not all tools, just a collection found most useful when developing my recent applications. Most times when a feature is hard to test, another developer has already released a plugin, so searching for it on google or github before writing endless lines of test-code saves time on development and maintenance. Countless plugins and gems are available; the best way to stay informed about the ever-changing world of tools is subscribing to some plugin or gem resources.

6. EXAMPLE TDD

6.1 INTRODUCTION

Since only reading about a problem is often very abstract, I will now build an example application, first using traditional `Test::Unit` with normal functional tests, halfway through convert to `RSpec` and finish by separating functional tests into Controller and View tests with support from `Mocha`. This provides a first-hand experience with most of the tools from chapter 5 and will help to clarify when and where they can be used and how they play together to thoroughly test an application.

The application created will by no means be a complete website, but a typical section of it, Model View and Controller of one resource that is handled by the application. Since most applications have to deal with users in some way or another the example will focus on creation and management of users.

6.2 BASIC USER TESTS

To make thing fast and short, I rely mostly on scaffold code and only test things not generated by rails. The basic setup for this application is generated by the rails command and the scaffold generator:

```
$ rails tdd-example && cd tdd-example
$ script/generate scaffold user
#change config/routes.rb (map.root :controller => "users")
#delete public/index.html
```

Before doing anything, build some basic unit tests, to sort out what the model needs. Install `Single Test` and `valid_attributes` and then continue by filling `test/unit/user_test.rb`:

```
class UserTest < ActiveSupport::TestCase
  fixtures :users
  def setup
    @user = users(:one)
  end

  it 'valid attributes should create valid user' do
    create_valid User
  end

  it 'should find invalid attributes' do
    assert_invalid_attributes User, :login => ['', nil], :email => [nil, '', 'ohno']
  end

  it 'should not allow duplicated fields' do
    @user = create_valid User
    assert_invalid_attributes User, :login => @user.login, :email => @user.email
  end
end
```

When we trying to see the tests fail, rails warns that the database migrations have to be run first:

```
#db/migrate/001_create_users.rb
create_table :users do |t|
  t.string :email, :login, :null => false
  t.timestamps
end
```

```
$ rake db:migrate
$ rake test:user #see 5.6.3 or use rake test:units
```

Test results: 6 tests, 5 assertions, 5 failures, 0 errors

To satisfy the tests modify app/models/user.rb:

```
class User < ActiveRecord::Base
  validates_presence_of :email, :login
  validates_format_of :email, :with => /@/
  validates_uniqueness_of :email, :login
end
```

Rerunning the tests now results in: 6 tests, 5 assertions, 0 failures, 0 errors

6.3 BASIC CONTROLLER TEST

Now I turn to the controller, install form_test_helper 5.3.5 and optionally railstidy 5.3.6 / html_test 5.3.7.

When running the generated controller tests, they result in 7 tests, 11 assertions, 1 failures, 0 errors.

Investigation shows that create fails since the model validations were already changed. The tests for a successful save or create therefore need a set of valid attributes when being called. After this is fixed, I can define scenarios where validations fail and result in a different template being rendered:

```
def test_create_should_fail_when_data_is_invalid_and_show_new
  get :new
  submit_form :user => { :email => 'something@web.de', :login=>'' }
  assert_template 'new'
end
#similar for update
```

After these additions controller test results are 9 tests, 17 assertions, 0 failures, 2 errors, both failures are 'Field named 'user' not found in FieldsHash', meaning that the user form has no room for login and email.

These two failing tests can be made to pass by building the appropriate user-form:

```
app/views/users/edit.html.erb and new.html.erb:
<% form_for(@user) do |f| %>
  <%= render :partial => 'form', :locals =>{:f => f}%>

app/views/users/_form.html.erb
<fieldset>
  <label for="user_login">Login</label> <%= f.text_field 'login' %> <br/>
  <label for="user_email">Email</label> <%= f.text_field 'email' %> <br/>
</fieldset>
```

After this controller tests are 9 tests, 23 assertions, 0 failures, 0 errors.

6.4 CONVERTING TO RSPEC

Now I install RSpec (5.4) and RSpec Converter (5.4.2) then convert all tests to RSpec with the RSpec Converter. The converted tests can be run with `rake spec` and result in **15 examples, 0 failures**. There are not 23+6 examples since RSpec only counts each test case, unlike Test::Unit which counts every assertion.

Now I will delete all old tests, to not overwrite the generated specs in case I happen to run `convert_to_rspec` again. Then I will have a look at the specs and refactor anything that could be enhanced:

```
assert_difference('User.count', -1) do
  delete :destroy, :id => users(:one).id
end
  ➔ in create and delete
lambda {
  delete :destroy, :id => users(:one).id
}.should change(User, :count).by(-1)
```

```
assigns(:user).valid?.should_not be_true
  ➔ in many examples, makes our intent clearer
assigns(:user).should_not be_valid
```

6.5 SPLITTING THE CONTROLLER

Splitting the `users_controller_spec.rb` into controller, view tests and stories will be the next step. For this Mocha 5.3.8 is needed and in `spec/spec_helper.rb` the line `# config.mock_with :mocha` must be uncommented.

I do not want to decrease test coverage and keep everything running, so I will work on the view tests first, then the stories and finally remove any view/integration related testing from the controllers.

6.5.1 VIEWS

The Rspec controller generator can also be used to generate view tests. With this command a test for the users index view is created: `script/generate rspec_controller users index`

The generated file in `spec/views/users/index.html.erb_spec.rb` is almost empty and will now be filled by a basic view test that checks if all users supplied by the controller were rendered:

```
require File.dirname(__FILE__) + '/../../spec_helper'

describe "/users/index" do
  fixtures :users

  before(:each) do
    assigns[:users]=[users(:one),users(:one)]
  end

  it "sould have a row for each item" do
    render '/users/index'
    response.should have_tag('#user_listing tr', :count=>2+1) #2items + heading
  end
end
```

Running this test with `rake spec:views` results in **1 example, 0 failures**.

6.5.2 STORIES

See RSpec:Running Stories 5.4.1 for getting started help.

The story runner lies in `stories/run.rb`, is executed with `'ruby stories/run.rb'` and contains:

```
require File.join(File.dirname(__FILE__), "helper")
require File.join(File.dirname(__FILE__), 'form')

with_steps_for :users, :general do
  run_story 'users'
end
```

For more advanced form interaction using the `form_test_helper` plugin 5.3.5, I created `stories/form.rb`:

```
#re-selecting the form means loosing field-data
def form(selector=nil)
  @form ||= select_form selector
end

#reset the form object on each submit
def form_submit
  @form.submit
  @form = nil
end

#enter a nested value into the form
def form_enter(type, valid_attributes)
  valid_attributes.each { |field,value| form.send("#{type}[#{field}]",value)}
end
```

The actual user story lies in `stories/stories/users.txt`:

```
Story: User interaction
  Scenario: protect from creating a user without email
    when i go to users/new
    And i enter valid attributes except email
    And i submit
    Then i should stay at new
    And the user should have an error on email

  Scenario: create a new user
    when i go to users/new
    And i enter valid attributes
    And i submit
    Then i should be redirected to the user
```

Now that everything is set up the story can be executed. At the moment only a reminding `'(PENDING)'` can be seen behind each line of the story, which is the RSpec way of saying "this step is not built yet".

There are 2 kinds of 'sentences' in this story, Then and when, all 'And' sentences are the same type as the one before them. To finally be able to run the stories, all used sentences must be covered by a corresponding step defined inside the in `stories/steps/users.rb`:


```

valid_attributes = {'login'=>'newuser', 'email'=>'new@email'}

steps_for(:users) do
  when("i enter valid attributes") do
    form_enter 'user', valid_attributes
  end

  when("i enter valid attributes except $attribute") do |attribute|
    form_enter('user', valid_attributes)
    form.user[attribute]=''
  end
end

```

These are of course not all steps that the story need, only the user-related ones. All steps that are very general and can be reused by other stories in the application, will be placed in `stories/steps/general.rb`.

```

steps_for(:general) do
  when("i go to $url") do |page|
    get page
  end

  when("i submit") do
    form_submit
  end

  Then("i should stay at $url") do |url_or_action|
    response.should render_template(url_or_action)
  end

  Then("the $record should have an error on $field") do |record,field|
    assigns(record).errors.on(field).should_not be_nil
  end

  Then("i should be redirected to the $record") do |record|
    response.should redirect_to(assigns(record))
  end
end

```

With 7 steps, from which only 2 are ‘user’ centered, 9 story sentences where covered. This is a good reuse quota and a sign that story testing can be repetition free, when steps are built to be reusable.

Running the story now succeeds without pending steps.

6.5.3 MOCKING IN THE CONTROLLER

The form interaction is covered inside the stories, so it can be removed from the controller tests. I start mocking all database actions, so there is no longer a need to worry about whether the controller logic failed, or the model was not saved properly. Real “controller unit tests” without any view or database interaction. This is done by remove ‘integrate_views’ from tests, then defining general stubs using Mocha. The `valid_attributes` are no longer needed, since now it is not important if a model is valid or not.

All invocations of calls to `User` will be mocked inside the testcase that uses it, e.g. index:

```
User.expects(:find).with(:all).returns([])
```

For each action I test the success and failure route to see if the controller reacted appropriately. It is not tested if a record was saved or that the `User.count` was increased, since all database concerns are eliminated.

```
it 'destroy should delete record and redirect to users' do
  @user.expects(:destroy).once.returns(true)
  delete :destroy, :id => users(:one).id
  response.should redirect_to(users_path)
end

it 'create should build new User, then save and redirect to the new User' do
  expect_new
  expect_save true
  #without this user_path(@user) would return /users/
  @user.stubs(:new_record?).returns(false)

  post :create, :user => {}

  response.should redirect_to(user_path(@user))
end

it 'create should render new when failing to save' do
  expect_new
  expect_save false

  post :create, :user => {}

  response.should render_template('new')
end
```

The helpers `expect_new` and `expect_save` that simplify testing since they encapsulate an often needed stub, are stored in `spec/spec_helper.rb`:

```
def expect_new
  @user = User.new
  User.expects(:new).returns(@user)
end

def expect_save val
  @user.expects(:save).returns(val)
end
```

After the refactoring controller tests pass with the same result as before, but now they are true unit tests and will be much more resistant to changes in views or models.

6.6 STARTING FROM SCRATCH

Now a new feature is built truly test-first, as opposed to the last example where existing code was only extended and refactored. A new feature should be built on a use-case, for this feature it will be: "When users behave badly, it should be possible to deactivate them and reactivate them if necessary." This implies the following concrete features:

- A new user is activated by default
- A user should be deactivateable
- A deactivated user should be activateable
- Index should only show activated users

6.6.1 INTEGRATION TEST

From the given feature description I have built an RSpec integration test (a story), so that the fulfillment of the use-case can be automatically tested through the story runner.

```
Scenario: deactivating a user
  when I go to users
  Then I should not see the first user
  when I deactivate the first user
  And I go to users
  Then I should not see the first user
```

I did not use the whole form submission process, since it is already covered by other stories, therefore the steps "go to edit", "change the form" and "submit the form" are missing from this story. When the story is run the steps 2, 3 and 5 are marked as missing and must be built. These new steps go into `stories/steps/users.rb`, since they belong into the user world.

```
when("I deactivate the first user") do |field,value|
  User.first.update_attribute(:activated, false)
end

Then("I should not see the first user") do
  response.body.should_not =~ %r[users/#{User.first.id}[^\\d]/]
end
```

This new story should fail and it does: **3 scenarios: 2 succeeded, 1 failed, 0 pending**. But the stories are not yet finished since the users form submission test does not check if a user can actually set the activated attribute. This can be fixed by adding 'activated' => '1' to the valid attributes Hash inside the users form test, which then will verify that a field named `user[activated]` is present.

6.6.2 MODEL TESTS

The next step in fulfilling the new integration test is to alter the user model so that it stores activated and has a method that will fetch all currently activated users. But before building this the model needs unit-tests for these new features:

```
it 'should be activated by default' do
  User.new.should be_activated
end

it 'should only find activated' do
  create_valid User, :activated => false
  create_valid User, :activated => true
  User.activated.should_not be_empty
  User.activated.reject(&:activated?).should be_empty
end
```

These tests both fails since there is no column in the database called activated. Therefore a new migration that adds this column is needed.

```
script/generate migration add_activated_to_users

class AddActivatedToUsers < ActiveRecord::Migration
  def self.up
    #we have to add null=>false, or else it can be true,false and nil!
    add_column "users", "activated", :boolean, :default => true, :null => false
  end

  def self.down
    remove_column "users", "activated"
  end
end
```

After this migration was run (`db:migrate && rake db:test:prepare`) the first test already succeeds but the second still fails, so I add a named scope for activated to make it pass:

```
named_scope :activated, :conditions => { :activated => true }
```

After this addition the model is complete: 8 examples, 0 failures

6.6.3 CONTROLLER TESTS

Activated will only be another field in the user form, so the tests for create and save already cover it. To make sure that `User.activated` is used inside the index action, the controller tests for index needs to be changed:

```
User.expects(:activated).returns([123])
assigns[:users].should == [123]
```

This new assumption will fail: 9 examples, 1 failure and is made successful with a small addition to the users controller:

```
def index
  @users = User.activated
end
```

Running the controller tests shows everything passed: 9 examples, 0 failures.

6.6.4 VIEW TESTS

So far there is no test for the `_form` partial. All that I want to verify is that every model field has a corresponding form element and this is already tested through stories. To make the stories pass I fix the `app/users/_form.html.erb` by adding:

```
<label for="user_activated">Activated</label>
<%= f.check_box 'activated' %> <br/>
```

6.6.5 INTEGRATION TEST CONTINUED

After all unit tests have passed and the form was fixed, nothing stops the integration test from failing and they pass for the first time with **3 scenarios: 3 succeeded, 0 failed, 0 pending**.

6.6.6 RCOV

After all tests are satisfied it is time to check that all new code is covered by tests. This can be done by running `rake spec:rcov`, `rcov` gives no guarantee that every bit of logic is tested, but shows locations that miss testing. And since everything is covered by the new unit tests, `rcov` results are 100% green:



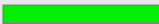
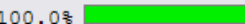


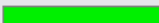
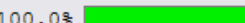


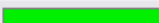
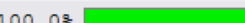
Name	Total lines	Lines of code	Total coverage	Code coverage
TOTAL	132	78	100.0% 	100.0% 
app/controllers/application.rb	10	4	100.0% 	100.0% 
app/controllers/users_controller.rb	97	60	100.0% 	100.0% 
app/helpers/application_helper.rb	3	2	100.0% 	100.0% 
app/helpers/users_helper.rb	2	2	100.0% 	100.0% 
app/models/user.rb	20	10	100.0% 	100.0% 

Figure 7: RCov results from example application test run

6.7 CONCLUSION

This sample section of a real application provides an overview of techniques and organization principles needed for testing. Extending it with new features or starting from scratch with a new application should be easy, once the initial blocker of “where to start” and the first setup problems are overcome. When only building more or less standard parts like the `UserController` shown, repetition may occur, that can be dried up with customized helpers (e.g. `assert_behaves_like_index`) or by using a library (e.g. `make_resourceful / resource_controller`), so that they only need testing when leaving the standardized path.

7. CONCLUSION

"There's an[...] quality about test-first design that gives you a sense of unhurriedness. You are actually moving very quickly, [...] because you are creating little micro-goals for yourself and satisfying them. At each point you know you are doing one micro-goal piece of work, and it's done when the test passes.[And] You just have to think about one little piece of responsibility. " (Fowler, Test-Driven Development, 2008)

Testing is integrated into Rails and an essential part of its architecture. As soon as one gets started with a new application there are test directories and structures just waiting to be filled, an invitation for every developer to start testing. Starting to test and using the included tools can be an enormous advance for developers that seldom tested before, but without testing guidelines and specialized tools these efforts can stay fruitless since they quickly become dull and repetitive. The right methods and tools can keep tests organized and dry, thus being easier to maintain and making the developer more productive.

With the Test-Driven approach code and code coverage can be improved without requiring more time for tests than before, since theoretically the same amount of tests would be written anyway. Testing first leads to better tested and loosely coupled code that is easy to maintain. Most of the shown tools and techniques can be used without test driven development, but doing so would be a waste of potential. The goal of these techniques is to build automate tests from model through controller and views, up until the Javascript, so that the whole application logic is covered with tests and developers can work and refactor without having to open the browser and to click through all scenarios.

8. LIST OF FIGURES

Figure 1: Comparisson of thick and thin client.....	4
Figure 2: ACID test - same CSS, different browser	8
Figure 3: Example of unobtrusive Javascript.....	30
Figure 4 Result of JUnit test run	31
Figure 5: Result of jQuery test run	32
Figure 6: Selenium IDE inside Firefox, actions are being recorded and can be enhanced through assertions directly in the browser	33
Figure 7: RCov results from example application test run	45

9. SOURCES

- Astels, D. (2006). *RSpec Cheat Sheet*. Retrieved 10 15, 2008, from <http://blog.daveastels.com/files/QuickRef.pdf>
- Beausoleil, F. (2008). *Piston*. Retrieved 02 12, 2009, from <http://piston.rubyforge.org/>
- Cauldwell, N. (2007). *Arts*. Retrieved 10 12, 2008, from <http://www.railsforum.com/viewtopic.php?pid=55887>
- Cohen, Larson, & Ware. (2001). Improving Software Investments through Requirements Validation. IEEE 26th Software Engineering Workshop.
- Erdogmus, Morisio, & M, T. (2009, 01 01). *On the Effectiveness of the Test-First* . Retrieved from http://iti-iti.nrc-cnrc.gc.ca/publications/nrc-47445_e.html
- Eric. (2006). *Test Rails*. Retrieved 06 12, 2008, from <http://www.verticalexpressionsoftware.com/blog/2006/06/17/converting-to-using-testrails/>
- Err. (2007). *evil-twin*. Retrieved 10 12, 2008, from <http://errtheblog.com/posts/67-evil-twin-plugin>
- Err. (2007). *test_spec cheatsheets*. Retrieved 02 12, 2009, from http://cheat.errtheblog.com/s/test_spec/
- Fowler, M. (2004). *dependency injection*. Retrieved 02 22, 2009, from <http://martinfowler.com/articles/injection.html>
- Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison Wesley.
- Fowler, M. (2008). *Test-Driven Development*. Retrieved 02 13, 2009, from <http://www.artima.com/intv/testdrivenP.html>
- Foz works. (2007). *Autotest sound*. Retrieved 10 12, 2008, from <http://www.fozworks.com/2007/7/28/autotest-sound-effects>
- Garber, J. (2008). *form_test_helper*. Retrieved 06 15, 2008, from http://continuous.rubyforge.org/form_test_helper/rdoc/
- Grosser, M. (2008). *AgileDox*. Retrieved 1 15, 2009, from <http://code.google.com/p/agiledox-rake/>
- Grosser, M. (2008). *Autotest notifications*. Retrieved 04 15, 2008, from <http://pragmatig.wordpress.com/2008/04/15/autotest-rspec-notifications-for-ubuntu/>
- Grosser, M. (2008). *jQUnit*. Retrieved 06 06, 2008, from <http://code.google.com/p/jqunit/>
- Grosser, M. (2008). *valid attr*. Retrieved 10 12, 2008, from <http://pragmatig.wordpress.com/2008/10/03/dry-validation-testing-and-easy-edge-case-records>
- Grosser, M. (2009). *valid attributes plguin*. Retrieved 1 15, 2009, from http://github.com/grosser/valid_attributes
- Hale, C. (2006). *Rails Rcov*. Retrieved 10 15, 2008, from http://blog.codahale.com/2006/05/26/rails-plugin-rails_rcov/
- Indeed. (2009). *Ruby*. Retrieved 2 9, 2009, from <http://www.indeed.com/jobtrends?q=ruby&l=&relative=1>
- McConnel, S. C. (2004). *Code Complete*. Microsoft Press.

Mead, J. (2008). *Mocha*. Retrieved 10 15, 2008, from <http://mocha.rubyforge.org/>

Meester, J. (2008). *RSpec Conversion*. Retrieved 10 15, 2008, from http://rspec.info/documentation/test_unit.html

Merenne, D. (2006). *page-cache-test*. Retrieved 10 12, 2008, from <http://blog.cosinux.org/pages/page-cache-test>

Merenne, D. (2006). *Rails tidy*. Retrieved 01 14, 2009, from <http://www.cosinux.org/~dam/projects/rails-tidy/doc/>

Preston-Werner, T. (2007). *fixture-scenarios*. Retrieved 10 12, 2008, from <http://code.google.com/p/fixture-scenarios/>

quircksmode.org. (2006). *Get Styles*. Retrieved 06 05, 2008, from <http://www.quircksmode.org/dom/getstyles.html>

Railstidy. (2008). *Railstidy quickref*. Retrieved 06 20, 2008, from <http://tidy.sourceforge.net/docs/quickref.html>

Rice, D. J. (2007). Retrieved 06 08, 2008, from <http://www.davidjrice.co.uk/articles/2007/8/12/ruby-on-rails-plugin-test-unit-to-rspec-converter>

RSpec. (2008). *RSpec home*. Retrieved 10 15, 2008, from <http://rspec.info/>

Sanheim, R. (2006). *BDD in Rails*. Retrieved 06 12, 2008, from <http://robsanheim.com/2006/12/29/bdd-in-rails-testspec-and-rspec/>

Sanheim, R. (2007). *Red Green*. Retrieved 10 13, 2008, from <http://robsanheim.com/2007/07/24/did-redgreen-get-borked/>

Scofield, B. (2008, 10 12). *Laziness*. Retrieved from <http://github.com/bscofield/laziness/tree/master>

script.aculo.us. (2006). *Test Runner*. Retrieved 12 10, 2008, from <http://wiki.script.aculo.us/scriptaculous/show/Test.Unit.Runner>

Selenium. (2008). *Help*. Retrieved 1 15, 2009, from <http://selenium-on-rails.openqa.org/>

Sunnyfortuna. (2006). *unit_tests*. Retrieved 12 10, 2008, from http://sunnyfortuna.com/scripts/test/run_unit_tests.html

Thomas, M. (2001). *IT Projects Sink or Swim*. British Computer Society Review.

ZenTest. (2006). *Test Rails*. Retrieved 10 12, 2008, from <http://zetest.rubyforge.org/ZenTest/classes/Test/Rails.html>