Developing in the Open
by Jeremy Friesen

Developing in the Open

One Programmer's Take on Development

Jeremy Friesen

Contents

Preamble

This work has been a labor of love. I write it in my spare time.

If you are feeling generous please consider throwing a tip my way via Paypal jeremy.n.friesen@gmail.com or give a donation to the American Civil Liberties Union or Wikipedia.

I warn you, this work is not complete, but I hope there is useful information even in its gestational state. This is something I feel needs to get out and into the open.

Want to help me make this better? Submit a pull request.

Preface

Since early in my professional programming career, I have often worked alone.

Working alone began while working as an actuarial analyst. The company was small and unable to staff all of its analyst positions. I realized most of what was needed was automation. I ended up automating away the need for one of the positions that the company never staffed.

Working alone continued at my next job as each developer worked on a single project. I did end up mentoring a new developer, and we worked well together.

For my next job, there were a total of two developers stretched between system administration, server-side application development, up to client-side application development via javascript, HTML and CSS development. No worries, it was a major application for our employer.

And now I find myself working on a team that is comprised of people at my institution as well as members at several different institutions. My team is distributed: by location, by a multitude of projects, and varying hours of commitment per week.

I've learned a lot from the latest collaborations. First, I have habits that are hard to break; I struggle to be a member of the team. Second, I value face to face interaction, but want to retreat to my desk so I can get work done.

I have been tasked with being the technical lead. It has been an occassion for personal growth.

I'm discovering that I have an arsenal for shooting most kinds of trouble. Inside this book you will find some heuristic tools, advice, and even conjecture.

Are these the best ways to be doing things? Maybe, maybe not. But it is my take on developing in the open.

Status: @READY_FOR_QA

Chapter 1

Command Line

When I'm not writing code in my text editor, I'm often on the command line. If you are new to the Linux/Unix ecosystem it may be intimidating. I encourage you to spend time getting to know your command line.

1.1 Useful Shell Commands

I am comfortable on the command line, but am not an expert. In fact one of the reasons I like Ruby so much is that it integrates quite nicely with the command line.[1]

Below is a small primer of only a handful of functions. For more detail, I'd recommend seeking out other resources. An excellent Bash cheatsheet can be found at learncodethehardway.org

1.1.1 Getting your bearings

The command line can be an initially disorienting location. The following three commands are the basics for navigation: **`pwd`**, **`ls`**, and **`cd`**.

[1]One of the things I love about Ruby is how seamlessly it integrates with the command line. David Copeland's Build Awesome Command-Line Applications in Ruby 2 is one resource.

pwd - where am I at?

```
$ man pwd

The pwd utility writes the absolute pathname of the current working directory to
the standard output.
```

Find out the path to where you are on the file system.

cd - change where I am at

```
$ cd path/to/another/directory
```

Change directory.

ls - see what is around me

```
$ man ls

List directory contents
```

If you want to see information about files in your working directory, a quick `ls` will tell you. You can provide an alternate directory.

which - where is the program file that is executing

```
$ man which

The which utility takes a list of command names and searches the path for each executable file
```

Out of the box your machine will come with lots of commands. You may configure your $PATH to prepend a directory for custom installs. After awhile you may not know which file is being call. `which` is here to help.

1.1.2 man

```
$ man man

format and display the on-line manual pages
```

If you have questions about any of the commands, try typing **`man`** and the name of the function. This brings up the manual for that command.

Some manuals are better than others.

1.1.3 grep

```
$ man grep

The grep utility searches any given input files, selecting lines that match one
or more patterns. By default, a pattern matches an input line if the regular
expression (RE) in the pattern matches the input line without its trailing
newline. An empty expression matches every line. Each input line that matches at
least one of the patterns is written to the standard output.
```

When I first started open source programming, I had never heard of Regular Expressions. The **`grep`** command changed that.

It is often the first tool I use for sifting through lots of files. There are lots of flavors of **`grep`** that I have barely explored (**`grep`**, **`egrep`**, **`fgrep`**, **`zgrep`**, **`zegrep`**, **`zfgrep`**).

Good luck, and tell me what you find out.

ack

```
$ man ack

Ack is designed as a replacement for 99% of the uses of grep.

Ack searches the named input FILEs (or standard input if no files are named, or
the file name - is given) for lines containing a match to the given PATTERN.
By default, ack prints the matching lines.
```

If you are using Mac OS X, you'll likely need to install ack or ag.

I haven't trained myself to use ack, but by many accounts it is more useful for current work.

1.1.4 sed

```
$ man sed

The sed utility reads the specified files, or the standard input if no files are
specified, modifying the input as specified by a list of commands.  The input is
then written to the standard output.
```

One of my early Rails projects was working with a massive XML document. There were approximately 30,000 top level nodes each with 30 or so child/grand-child nodes. It was impossible to load into a tree parser.[2]

I ended up writing a `sed` script that split the document into 30,000 lines. Each line representing a single top level node and its child nodes.

From there I was able to read each line, and use a tree parser to individually parse each top-level node. This solution was better than writing a REXML::StreamListener and worrying about tracking tag open and tag end.[3]

1.1.5 find

```
$ man find

The find utility recursively descends the directory tree for each path listed,
evaluating an expression (composed of the `primaries' and `operands' listed
below) in terms of each file in the tree.
```

[2]The tree parser's available at the time required that the entire XML document be read in one gulp. Each XML element would then be represented as one or more objects. So 30,000 top-level elements, and 30 or so child elements meant at least 900,000 XML specific objects would need to be created to represent that document, as well as the text and attributes. That is a ridiculous amount of objects to hold in memory.

[3]A stream listener reads the XML document character by character and essentially notifies the listener when you are opening a tag, closing a tag, encountering cdata, etc. Needless to say, to use a listener a developer must potentially track a lot of state.

A powerful command which I most often use for listing filenames that match a textual pattern.

```
$ find . -name my_ruby_file.rb
```

But wait, it can do so much more! Consider the following example provided in `find`'s man page.

```
# Delete all broken symbolic links in /usr/ports/packages

$ find -L /usr/ports/packages -type l -exec rm -- {} +
```

1.1.6 xargs

```
$ man xargs

The xargs utility reads space, tab, newline and end-of-file delimited strings
from the standard input and executes utility with the strings as arguments.
```

A simple example is perhaps best for explaining `xargs`. The following command `echo "hello.txt" | xargs touch` is analogous to `touch hello.txt`. Both commands will create an empty file named "hello.txt".

The following script:

```
$ find . -name *hello* | sed -e "p;s/hello/world/" | xargs -n2 mv
```

1. Finds all files and directories with names that contain "hello"
2. Replacing the "hello" portion of the name with "world"
3. Move the files and directories with names that contain "hello" to files (or directories) with the "hello" portion of the name replaced with "world".

I haven't tested the above script for all scenarios, but it has served me well when used in conjunction with source control.

1.1.7 Miscellaneous Commands

What follows are a handful of commands that I have stitched together. There are likely other ways of doing these things, but they are the solutions that I have found.

Change directory into the parent directory of an executable file

```
$ cd `which ruby`/..
```

The above command will find where your `ruby` executable is, then change into the containing directory of that executable. The backticks are for command substition. The `which ruby` is executed and the returned value is used in the context of the containing command. That is if `which ruby` returned `/path/to/ruby` then the above command would be equivalent to `cd /path/to/ruby/.`

Kill every instance of Jetty

For some reason, Jetty doesn't always stop. I went ahead and created a `kill-jetty` alias that executes the following:

```
$ ps ax | grep jetty | grep -v grep | sed "s/^ *//g" | cut -f1 -d " " | xargs kill -9
```

In writing this book, I learned that I could do the above with the very concise command:

```
$ pkill -f jetty
```

The above string of commands highlights how simple Unix commands can be sewn together to do powerful things.

Commits since master

Within a git repository, count the number of commits that your current branch is ahead of the master branch.

```
$ git log master..`git rev-parse --symbolic-full-name --abbrev-ref HEAD` --oneline | wc -l | tr
```

1.2 My `dotfiles` repository

The configuration files of my `$HOME` directory are as important as any code I write. It is both the tools and toolbox that I carry with me to do my job.

After reviewing various dotfile project's at dotfiles.github.io, I forked @holman's dotfile project. This meant switching the project from `zsh` to `bash`.

1.3 My `~/.inputrc`

First I make sure that my command line navigation is up to snuff. I export the following custom INPUTRC.

```
# http://www.cs.colostate.edu/helpdocs/emacs.html
set editing-mode emacs
set completion-ignore-case on

# Allow the command prompt to wrap to the next line
set horizontal-scroll-mode Off

# Allow Unicode
set meta-flag on
set input-meta on
set output-meta on
set convert-meta off


"\e[5~": beginning-of-history # Page Up
"\e[6~": end-of-history       # Page Down

# Mappings for <Ctrl>+<Left Arrow> and <Ctrl>+<Right Arrow> word navigation.
"\e[1;5C": forward-word
```

```
"\e[1;5D": backward-word
"\e[5C": forward-word
"\e[5D": backward-word
"\e\e[C": forward-word
"\e\e[D": backward-word

# These allow you to start typing a command and use the up/down arrow to auto
# complete from commands in your history.
"\e[B": history-search-forward
"\e[A": history-search-backward
```

To enable your `INPUTRC` add the following lines to your `~/.profile`.

```
if [ -f $HOME/.inputrc ]; then
  export INPUTRC="$HOME/.inputrc"
fi
```

The dotfiles project handles this as part of its `script/bootstrap`.

1.4 My `project` command

I work on a lot of repositories over the course of a week. The vast majority of them are Ruby repositories.

As a matter of convention, all of my repositories are in `~/Repositories/`. This convention and my `project` command means I can open my projects with minimal typing.

On the command line, if I type `project hydra`, then hit `tab` I get an tab completion list of matching directories in the `~/Repositories/` directory.

```
$ project hydra<tab>

hydra                          hydra-collections              hydra-jetty
hydra-account_manager          hydra-derivatives              hydra-jetty-cookbook
hydra-camp                     hydra-ezid                     hydra-migrate
hydra-camp-devise-testing      hydra-file_characterization    hydra-mods
hydra-remote_identifier        hydramata                      hydra-head
hydra-object_viewer            hydramaton
```

Or, if I type `project hydra`, then hit `enter` my text editor opens that directory. In reality, I create a Sublime project for that directory and then open that project; but to many it might look like I'm just opening the project.

Below are the three scripts that are necessary for the above behavior.

1.4.1 Code Sample for `project`

In `~/Repositories/dotfiles/bin/project`

@TODO - This is a rather lengthy file. A URL might be better served. Or an appendix entry?

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

project_name = ARGV[0]
if !project_name && ENV['PWD']
  project_name = ENV['PWD'].split("/").last
end

if project_name
  project_file_name = File.join(
    ENV['HOME'],
    'Projects',
    "#{project_name}.sublime-project"
  )
  repository_dirname = File.join(
    ENV['HOME'],
    'Repositories',
    project_name
  )

  unless File.exist?(repository_dirname)
    $stdout.puts %(Missing unable to find #{project_name.inspect})
    $stdout.puts %(  in #{File.dirname(repository_dirname)}")
    exit(-1)
  end

  unless File.exist?(project_file_name)
    # Creating the default project; Note this is very much configured for a
    # Rails project.
    File.open(project_file_name, 'w+') do |file|
      file.puts %({)
      file.puts %(  "folders":)
      file.puts %(  [)
      file.puts %(    {)
      file.puts %(      "path": "#{repository_dirname}",)
      file.puts %(      "folder_exclude_patterns": [)
      file.puts %(        ".yardoc",)
      file.puts %(        "pkg",)
      file.puts %(        "tags",)
      file.puts %(        "doc",)
      file.puts %(        "coverage",)
      file.puts %(        "tmp",)
      file.puts %(        "jetty",)
      file.puts %(        ".bundle",)
      file.puts %(        ".yardoc")
      file.puts %(      ],)
      file.puts %(      "file_exclude_patterns": [)
      file.puts %(        ".tag*",)
      file.puts %(        "*.gif",)
      file.puts %(        "*.jpg",)
      file.puts %(        "tags",)
      file.puts %(        "*.png",)
      file.puts %(        ".tags",)
      file.puts %(        ".tags_sorted_by_file",)
      file.puts %(        "*.log",)
      file.puts %(        "*.sqlite3",)
      file.puts %(        "*.sql")
      file.puts %(      ])
      file.puts %(    })
      file.puts %(  ],)
      file.puts %(  "settings":)
      file.puts %(  {)
      file.puts %(    "tab_size": 2)
      file.puts %(  })
      file.puts %(})
    end

    # Working with the file system, I may need to wait.
    sleep(1)
  end
  `subl #{project_file_name}`
  exit(0)
else
  $stdout.puts "Example `#{File.basename(__FILE__)} project_name`"
  exit(-1)
end
```

In `~/Repositories/dotfiles/project/completion.bash`

```
# project completion for Sublime projects by Jeremy Friesen
# <jeremy.n.friesen@gmail.com>
function _project {
  local cur
  COMPREPLY=()
  _get_comp_words_by_ref cur
  COMPREPLY=( $(find $REPOSITORIES \
  -type d -maxdepth 1 -mindepth 1 -exec basename {} \; | egrep "^$cur"))
}

complete -F _project -o default project
complete -F _project -o default p
```

Chapter 2

Git

One of the best tooling changes that I've experienced was transitioning from centralized version control to distributed version control.

At first it was the change to a tool that allowed fast branching and merging - a mandatory killer feature. Now it is the collaborative nature of the tool.

Box 2.1. I Merged Branches in SVN Once. Once.

Back in the early days of my Rails projects, I was working on a project managed in SVN. I was working on an extended feature and decided to create a branch. Work proceeded on both branches.

Then came the reckoning.

We attempted to merge the code. And everything fell apart.

We ended up spending 8 hours merging things. And I don't think we got it right. That day we decided we needed to move to git.

Did I mention we had minimal automated tests?

2.1 Bash Completion

I recommend that you enable **git**'s tab completion. Git has a large suite of commands. With tab completion installed, I've stumbled upon a few unknown commands and options.

Either the command:

```
$ git che<tab>

checkout      cherry        cherry-pick
```

Or the command's options:

```
$ git checkout --<tab>

--conflict=  --merge      --no-track   --orphan     --ours
--patch      --quiet      --theirs     --track
```

In your **~/.profile** add the following. *Or make use of my dotfiles repository 1.2.*

```
if [ -f /path/to/your/git-completion.bash ]; then
  source /path/to/your/git-completion.bash;
fi
```

Find out more about Git's bash completion.

2.2 Command Line Status

```
~/Repositories/take_on_rails (master *=) 2.0.0-p353 $
```

First is the path to the current directory (i.e. **pwd**).

```
~/Repositories/project
```

Then, if I am in a directory that is part of a Git project, the command prompt includes the current branch and repository status information.

```
(master *=$)
```

Output	What is going on?
`*`	My repository is in a "dirty state".
`$`	I have "stashed" something.
`=`	My branch is up-to-date with 'origin/master'.
`<`	My branch is behind 'origin/master'.
`>`	My branch is ahead of 'origin/master'.
`<>`	My branch is ahead of 'origin/master'.

I enabled these indicators by setting the following in my `~/.profile`:

```
GIT_PS1_SHOWDIRTYSTATE=true
GIT_PS1_SHOWSTASHSTATE=true
GIT_PS1_SHOWUNTRACKEDFILES=true
GIT_PS1_SHOWUPSTREAM="auto"
```

For more information you can review git's bash completion script.

```
2.0.0-p353
```

Display the current Ruby version. *This was more helpful a few years ago when I had a mix of 1.8.x projects and 1.9.x projects.*

Box 2.2. About My PS1

Below is a recent change to my PS1 script. I had different information, but I opted to instead use a community maintained set of scripts.

Working on lots of projects in various Ruby versions, I find this PS1 information vital. In fact, when I look at other people's PS1 and don't see git information, I am left wondering how they have room in their head to keep track of that information.

```
# Make sure git-prompt is available, otherwise this PS1 change will be very
# boring.
if [ -a `brew --prefix`/etc/bash_completion.d/git-prompt.sh ]; then

  GIT_PS1_SHOWDIRTYSTATE=true
  GIT_PS1_SHOWSTASHSTATE=true
  GIT_PS1_SHOWUNTRACKEDFILES=true
  GIT_PS1_SHOWUPSTREAM="auto"
  source `brew --prefix`/etc/bash_completion.d/git-prompt.sh

  # No way of knowing for certain if rvm-prompt is available.
  __rvm_ps1() {
    if [ -a $HOME/.rvm/bin/rvm-prompt ]; then
      echo " $($HOME/.rvm/bin/rvm-prompt v p g) "
    else
      echo ' '
    fi
  }

  # via http://tammersaleh.com/posts/a-better-rvm-bash-prompt
  bash_prompt() {
    local NONE="\[\033[0m\]"    # unsets color to term's fg color

    # regular colors
    local K="\[\033[0;30m\]"    # black
    local R="\[\033[0;31m\]"    # red
    local G="\[\033[0;32m\]"    # green
    local Y="\[\033[0;33m\]"    # yellow
    local B="\[\033[0;34m\]"    # blue
    local M="\[\033[0;35m\]"    # magenta
    local C="\[\033[0;36m\]"    # cyan
    local W="\[\033[0;37m\]"    # white

    # emphasized (bolded) colors
    local EMK="\[\033[1;30m\]"
    local EMR="\[\033[1;31m\]"
    local EMG="\[\033[1;32m\]"
    local EMY="\[\033[1;33m\]"
    local EMB="\[\033[1;34m\]"
    local EMM="\[\033[1;35m\]"
    local EMC="\[\033[1;36m\]"
    local EMW="\[\033[1;37m\]"

    # background colors
    local BGK="\[\033[40m\]"
    local BGR="\[\033[41m\]"
    local BGG="\[\033[42m\]"
    local BGY="\[\033[43m\]"
    local BGB="\[\033[44m\]"
    local BGM="\[\033[45m\]"
    local BGC="\[\033[46m\]"
    local BGW="\[\033[47m\]"

    local UC=$W                  # user's color
    [ $UID -eq "0" ] && UC=$R    # root's color

    PS1="$G\w$R\$(__git_ps1)$B\$(__rvm_ps1)${NONE}$ "
  }

  bash_prompt
  unset bash_prompt
fi
```

2.3 Git Config

I make extensive use of git aliases, to help encapsulate common behavior for my workflow. In my `~/.gitconfig` I've added a several aliases. And when using Bash Completion, these alias will also tab complete. Here are a few of them:

2.3.1 Show Very Verbose Branch Details

Compare the three examples:

Local branches with little information:

```
$ git branch
* chore-verify-isolation-tests
  master
```

Local branches only but with tracking information:

```
$ git branch -vv
* chore                       402150b [jeremyf/chore: ahead 1] Adding documentation
  master                      a7dda4a [origin/master] Merge pull request #1 from jeremyf/dltp-123
```

All branches with tracking information:

```
$ git branch -avv
* chore                       402150b [jeremyf/chore: ahead 1] Adding documentation
  master                      a7dda4a [origin/master] Merge pull request #1 from jeremyf/dltp-123
  remotes/jeremyf/chore       402150b Verifying specs run in isolation
  remotes/jeremyf/master      4cef983 Updating README with badges [skip ci]
  remotes/origin/master       a7dda4a Merge pull request #1 from jeremyf/dltp-123
```

At a glance, **`git branch -avv`** gives me the best perspective of what is happening.

I use **`git branch -avv`** all of the time, and went ahead and made **`git ba`** an alias. To do this, I added the following to my **`.gitconfig`**:

```
[alias]
        ba = branch -a -vv
```

@TODO

2.4 Bisect

From **`git-bisect`**'s man page.

> git-bisect – Find by binary search the change that introduced a bug

This is one of many killer features for **`git`**.

This is updated from a previous blog post.

2.4.1 Prepare Your Bisect

```
$ git bisect start
$ git bisect bad
$ git bisect good 3f5ee0d32dd2a13c9274655de825d31c6a12313f
```

- Tell git we are starting a bisect.
- Then tell git at what point we found the bug - in this case HEAD.
- And finally tell git at what point we were bug free – in this case at commit 3f5ee0d32dd2a13c9274655de825d31c6a12313f.

2.4.2 Run Your Test

We've told git-bisect the good and bad commits.

```
$ git bisect run ./path/to/test-script
```

And now we step through history and run `./path/to/test-script`.

What is `./path/to/test-script`? It is any executable file that exits with a 0 status (i.e. success) or a non-0 status (i.e. failure). Learn more about git-bisect run.

If `./path/to/test-script` is successful, the current commit is good. Otherwise it is bad. Git-bisect will converge on the commit that introduced the bad result and report the bad commit log entry.

2.4.3 Script Use Cases

What is this `./path/to/test-script`?

I've used `rake` for my Rails project. But that was overkill - my `rake` test suite was very slow.

I've also ran bisect on a test file in the repository (i.e. `ruby ./test/unit/page_test`
In these cases the test was in my the repository. This meant the test was subject to changes over the commit time range.

I have found using a script not in the repository to be the best option. I can answer the very specific question by using a custom script.

This was useful when my manager asked When did this strange behavior get introduced?

I wrote a Capybara test that automated the steps my manager reported. Reproducing the error and creating an assertion of the expected behavior.

A few weeks prior, I had introduced the odd behavior as a side-effect. At the time I didn't have test coverage for that particular behavior.

I patched the error, answered my managers question, and had an automated test that I could drop into my repository to make sure I didn't reintroduce that behavior.

2.5 When to Commit

I like to commit early and often. One trick I've used is to create a feature branch then make numerous commits, often times with terse messages. By doing this I have an excellent "undo" buffer.

Once I'm ready to share these changes, I will use an interactive rebase to amend the commit messages. This way I can focus on two distinct modes of concentration:

- Writing supporting specs and code
- Writing commit messages

For me, the mental state for those two modes is different. So I use the rebase trick to force my brain into writing more details concerning any of the changes. This may mean that two or more commits are squashed together as they might make sense together.

2.6 Git Immersion

@TODO - go through gitimmersion to speak authoritively

"Git immersion is a guided tour that walks through the fundamentals of git, inspired by the premise that to know a thing is to do it."
- *gitimmersion.com*

Chapter 3

Ruby

First, write your obligatory Hello World program. I'll help you out so we can proceed:

```
$ ruby -e 'puts "Hello World"'
```

Done!
Now on to other things.

3.1 Idioms

When I started in Ruby, there were three foreign concepts: blocks, modules mixins, and method missing behavior. I'll add regular expressions, as I was just starting to tinker with them.

3.1.1 Instance Method vs. Class Method Annotation

Consider the following class. You want to write about that class in a commit message, email, or IRC.

```
class MyClass
  def self.a_method
    'class method'
  end
  def a_method
    'instance method'
  end
end
```

To write about the class method, use `MyClass.a_method`. To write about the instance method, use `MyClass#a_method`.

3.1.2 Blocks

With the rise of Javascript, the idea of blocks is less obtuse. They can be thought of as anonymous functions.

In other words, when you pass a block as part of your message, the receiver decides how to handle the block. It could:

- Ignore it
- Call it 1000 times
- Pass it on as lambda to another method invocation

As with all functions, blocks take parameters. The `#each` method is a common receiver of a block, yielding one parameter.

```
[1,2,3,4].each do |index|
  puts index
end
```

The above Ruby code will write to STDOUT 1,2,3, and 4; Each separated by a new line.

You can also capture the block, storing it in an instance variable, and later executing that block. This is a very powerful tool for configuration management. More on that in the Rails section.

Warning Every method can take a block. But not every method will yield to the block it received. There have been plenty of times where I've typed the equivalent of `[1,2,3,4] {|i| puts i }` and never seen any output. You have been warned.

Then again, it can be helpful to not yield . Or yield multiple times. Its up to the receiver of the block to determine what to do with it.

Consider the following example:

```
request { |response|
  response.success { send_report }
  response.failure { retry }
}
```

We do not want both `send_report` and `retry` to be executed each time a request is made.

3.1.3 Module Mixins

Module Mixins are a powerful tool for object composition. But be careful, as you can craft a nightmare object by mixing in so many modules.

Module Mixins trample on encapsulation. As I've worked with Ruby I've started associating Module Mixins as an intermediary step in refactoring.

I find common behavior amongst two or more objects. I tease apart the common behavior, putting that behavior in a module.

I test the module in isolation. Then mix the module back into the associated parent classes.

Often, I will stop there. But the decision to stop does not come without cost.

Modules are horrible at encapsulation. They can trample on other methods of the containing class. And it is difficult to keep the module isolated from other aspects of the containing class.

But they are powerful. And pervasive in the Ruby (on Rails) ecosystem. Understand them.

3.1.4 Method Missing

Ruby exposes a powerful, yet easy to abuse tool: `method_missing`.

If you implement `method_missing` don't forget to:

- implement the corresponding `respond_to_missing?`.
- capture the method arguements
- capture any block that might be associated with the call

Failure to account for the above caveats can result in very difficult code to test.

At one point in 2011, I found a nasty bug in ActiveRecord::Base#method_missing. I ended up spending several hours tracking it down. Then nursing a monkey patch in our application until my pull request was part of a released version of Rails.

3.1.5 Regular Expressions

3.2 Interactive Ruby Shell (irb)

```
$ man irb

irb is the REPL(read-eval-print loop) environment for Ruby programs.
```

Back to the Hello World example.

```
$ irb
>> puts 'Hello World'
Hello World
=> nil
>>
```

3.2.1 Source Location

This is a helfpul trick:

```
obj = Object.new
obj.method(:method_name).source_location
=> ["/path/to/ruby/file.rb", "LINE_NUMBER"]
```

3.2.2 My `~/.irbrc`

A few helpful tweaks for IRB:

```
#!/usr/bin/ruby
require 'rubygems'
require 'pry'
Pry.start
exit
```

Or if you prefer, replace the default IRB behavior with Pry.

```
require 'rubygems'
require 'pry'
Pry.start
exit
```

3.3 Pry

> "Pry is a powerful alternative to the standard IRB shell for Ruby. It features syntax highlighting, a flexible plugin architecture, runtime invocation and source and documentation browsing." - *pryrepl.org*

I started using Pry in the summer of 2014. I had trouble rebuilding Byebug for my debugger. I opted to give Pry a try.

Based on how I had used Byebug, out of the box Pry had enough features for me to hobble along. Then I installed `pry-debugger` and I found myself no longer needing `byebug`.

It is a compelling replacement for IRB and Byebug. I am making a comittment to practice using Pry and exploring more of it.

Launch **pry** from the command line then walk along:

```
# See the methods for the current context
pry(main)> ls

self.methods: inspect  to_s
locals: _  __  _dir_  _ex_  _file_  _in_  _out_  _pry_

# Show information about in FileUtils
pry(main)> ls FileUtils
constants: DryRun  Entry_  LOW_METHODS  LowMethods  METHODS  NoWrite  OPT_TABLE  StreamUtils_
FileUtils.methods:
  cd        chown              commands         copy_entry   cp_r           install
  ...(And more methods)
instance variables: @fileutils_label  @fileutils_output

# Change context to the FileUtils module
pry(main)> cd FileUtils
pry(FileUtiles)> show-method rm

From: /Users/jfriesen/.rvm/rubies/ruby-2.1.2/lib/ruby/2.1.0/fileutils.rb @ line 562:
Owner: #<Class:FileUtils>
Visibility: public
Number of lines: 10

def rm(list, options = {})
  fu_check_options options, OPT_TABLE['rm']
  list = fu_list(list)
  fu_output_message "rm#{options[:force] ? ' -f' : ''} #{list.join ' '}" if options[:verbose]
  return if options[:noop]

  list.each do |path|
    remove_file path, options[:force]
  end
end
```

Great stuff. And **pry-debugger** introduces setting dynamic breakpoints and other such goodies.

3.4 Byebug

@TODO - A Ruby debugger

Chapter 4

Rubygems

Gems can be used to extend or modify functionality in Ruby applications. Commonly they're used to distribute reusable functionality that is shared with other Rubyists for use in their applications and libraries. Some gems provide command line utilities to help automate tasks and speed up your work. – guides.rubygems.org

If you spend anytime working in Ruby, you will make use of Gems. Rails is distributed as a gem.

The Gem Guides are a solid place to start.

4.1 Ruby-Toolbox.com

This is a goto resource for vetting gems. For a given category what are the usage and maintenance stats for those gems?

4.2 Gemnasium.com

Keep tabs on what gems are out of date.

4.3 Making Your Own Gems

If you are working on something that could be shared, consider making it a gem. In fact, many institutions make gems to share amongst their teams or projects.

I recommend using Bundler's `bundle gem <NAME>` command to create your own gems. I go into (more detail about the `bundle gem`)[/bundler#sec-bundle-gem] command in the Bundler chapter.

Status: @READY_FOR_EXTERNAL_REVIEW

Chapter 5

Bundler

"Bundler maintains a consistent environment for ruby applications. It tracks an application's code and the rubygems it needs to run, so that an application will always have the exact gems (and versions) that it needs to run." *bundler.io*

Once upon a time, in the dark days of yore, I was working on two Rails projects at the same time. Bundler did not yet exist. The pain of dependency management was intense. Gems were a shared jumbled mess.

Synchronizing gems amongst different developers was one challenge. Another was ensuring the gem setup was correct on the various production related environments.

If I were to develop just one project on my machine I wouldn't need bundler. But it is a part of the Ruby ecosystem now. Even if you work on one project, Bundler is easy enough to use to be worth the overhead. And I can assure you that you will work on more than one project on your machine.

5.1 Gemfile

A Gemfile lists the explicit dependencies of a Ruby project. The Gemfile.lock[1], if one exists, lists the resolved dependency graph, down to the specific version of each gem in the dependency graph.

If you have a Gemfile but not a Gemfile.lock, running `bundle` in the project's root directory will generate a Gemfile.lock for the project. The resulting Gemfile.lock is the evaluated dependency graph *for your machine*.[2] If someone else were to run `bundle` against the same Gemfile their resulting Gemfile.lock might vary.

However, once the Gemfile.lock is created, running `bundle` will install the same things[3].

Your Gemfile can reference gems that are:

- Up on Rubygems.org
- Are a particular branch on Github
- Located on your local file system
- Or any other gem source that you register

5.2 Bundle Show

```
$ bundle help show

Show lists the names and versions of all gems that are required by your Gemfile.
Calling show with [GEM] will list the exact location of that gem on your machine.
```

[1]If you are working on your own gem, make sure to not checking in your local Gemfile.lock. [More from Yehuda Katz, creator of Bundler can be found at http://yehudakatz.com/2010/12/16/clarifying-the-roles-of-the-gemspec-and-gemfile/.

[2]Bundler evaluates each declared gem and its dependency restrictions (i.e. `gem "rails", ">= 4"`, `rspec, "~>1.2"`, etc.).

[3]It is still possible that two machines with the same Gemfile.lock may have different dependency graphes. This is because Gemfile's can have imperatives for OS level differences.

Of the two options – with or without a gem name - I more often use `bundle show <name_of_gem>`.[4] I want to know where a gem was installed more often than I want to see the gem list.

And I often follow with `bundle open <name_of_gem>`.

5.3 Bundle Open

```
$ bundle help open

Opens the source directory of the given bundled gem
```

When I want to dig around in the gem's executed code, I run `bundle open <name_of_gem>`. My text editor pops open and I begin the spelunking; All of the gem's code is there to review.[5]

Sometimes I'll bring a pickaxe to the excavation and start modifying the code; Insert a debugger statement in one place[6]; Log parameters and collaborators in another location.

All of which is made easy by `bundle open <name_of_gem>`.

5.4 Bundle Console

```
$ bundle help console

Opens an IRB session with the bundle pre-loaded
```

[4]When I do use `bundle show <gem_name>` it is almost always followed by `bundle open <gem_name>`. Other times I am verifying the file system location of a gem that was declared with a `:path` or `:github` option (i.e. `gem "orcid", path: "../orcid"` or `gem "mappy", github: "jeremyf/mappy"`).

[5]Well not all of it. Its possible via the gemspec declaration that not all files were declared. Some people don't include the test or spec directory in their gemspec. Not cool! Personally I expect anything and everything that may be relevant to the workings of the code (and is not generatable) to be included.

[6]There have been a few times when I've altered my installed Gem and forgotten to clean it up. Usually this means I spend a bit of time re-opening the project and restoring things. A smarter person might wrap the `bundle open` command in initializing a git repository for the gem. Having waited for the close response from the editor, when `bundle open` quits reverting any changes. That may be magical overkill, so for now I'll settle on my error prone human ways.

I learned about this command late in 2013; Its been around longer than that, but I was a little dense in my failure to notice the command[7].

I prefer to stay out of the console and instead rely on tests, but it is another tool for exploring how your code is behaving.[8]

5.5 Bundle Outdated

```
$ bundle help outdated

Outdated lists the names and versions of gems that have a newer version
available in the given source. Calling outdated with [GEM [GEM]] will only check
for newer versions of the given gems. Prerelease gems are ignored by default.
If your gems are up to date, Bundler will exit with a status of 0. Otherwise,
it will exit 1.
```

As an Open Source Developer, your dependency landscape is shifting beneath you. Pay attention to changes in your dependencies.[9]

5.6 Bundle Update

```
$ bundle help update

Update  the  gems  specified  (all  gems,  if  none  are  specified), ignoring
the previously installed gems specified in the Gemfile.lock. In general, you
```

[7]It would behoove you, as a developer, to on occassion request system help for the scripts you regularly run. Somewhat randomly reading through the output of `bundle -h` in 2013 brought the awesome `bundle console` to my attention. Consider other commands you might use a lot: `rails`, `rake`, `rspec`, `git`, etc. Its software and subject to change. Be aware of those changes and occassionally sharpen your understanding of your tools.

[8]If you find yourself in a console "exploring things" please consider codifying that exploration into a test. Please capture your ad hoc exploration's "notes" for future spelunkers and code delvers. Once you have the test make sure your commit message explains what was happening. Trust me, your future self and those working on your project later will appreciate your carefully considered notes.

[9]Within a given gem, if a particular file has lots of churn over time, that is indicative of instability and volatility of the understanding of that file. Perhaps it is a misunderstood object, or an object that encapsulates changing business logic, etc. Regardless, you should be mindful of that. It is a larger liability than a stable file. My conjecture extends this from a file to a gem. If the gem is rapidly incrementing versions, you have a more volitale gem and should consider the ramifications. And yes, I'm guilty of rapidly versioning objects. @CITATION_DESIRED

```
should use `bundle install` to install the same exact gems and versions across
machines. You would use bundle update to explicitly update the version of a gem.
```

This removes the Gemfile.lock and generates a new one. It may create an identical Gemfile.lock (if you run `bundle update` back to back it has a high probability of being identical). If a lot of time has elasped between updates at least one dependency will have changed.

Before I run `bundle update` I do my best to make sure:

- that my tests are all passing
- that I have a clean working directory

After I've run `bundle update` I again do my best to make sure:

- that my tests are all passing
- that I understand what has changed amongst the dependencies

Assuming I am satisfied with my preliminary inspection I then stage and commit only the Gemfile.lock, commenting that I ran `bundle update`.[10]. I rely on git and the focused commit to ensure that I can revert my Gemfile.lock; Passing tests never guarantee that something works. This is most evident I update a Gemfile.lock and issues are found a day or so later.

Also consider the more targeted `bundle update <gem_name(s)>`. This command will keep all explicit locks except for the named gem(s). The named gem(s) will be updated along with those gem's dependencies.

5.7 Bundle Gem

[10]Someone, I don't recall who, recommended that when I run a script that alters the working directory (i.e. `bundle update` or `rails generate model ChickenPie`) that I should 1) make sure the working directory is clean before I run the script. 2) commit the changes made by the script. 3) For the commit message I should record the exact command(s) and options that was run. In following this procedure you are being a good citizen as you introduce code. A generator can create a tremendous amount of code. Keeping the scope of the commit to a single concept is being a good steward of your source control.

Status: @READY_FOR_QA

Chapter 6

Rake

One of the late Jim Weirich's pervasive contributions to the Ruby community. For many it is synonimous with "the script that run's my Rails project's tests."

It is more than that. It is a powerful build program.

```
$ man rake

Rake is a simple ruby(1) build program with capabilities similar to the regular make(1) command

Rake has the following features:

o   Rakefiles (Rake's version of Makefiles) are completely defined in standard Ruby syntax.  No
o   Users can specify tasks with prerequisites.
o   Rake supports rule patterns to synthesize implicit tasks.
o   Flexible FileLists that act like arrays but know about manipulating file names and paths.
o   A library of prepackaged tasks to make building rakefiles easier.
```

As with any of my tools, I look at the help on occassion. Rake has a few switches that are worth noting.

6.1 rake -W

Invoke rake with the `-W` option to see where each task is defined.

```
$ rake -W

rake install     ./bundler-1.5.2/lib/bundler/gem_helper.rb:43:in `install'
rake load_app    ./railties-4.0.3/lib/rails/tasks/engine.rake:1:in `<top (required)>'
rake release     ./bundler-1.5.2/lib/bundler/gem_helper.rb:48:in `install'
```

Knowing about this option can help you understand what the rake tasks you run are doing. It can also point you in the direction of how to write Rake tasks.

6.2 The Rake Field Manual

My appreciation for Rake continues to grow. Fanned ever higher by Avdi Grim's "Learn advanced Rake in 7 episode" and his upcoming "Rake Field Manual".

I'm not going to dive any further into this than to say, take a look at Avdi Grim's resources. He's built Quarto - an ebook generation toolchain powered by Rake.

Chapter 7

Rails

7.1 Fundamentals

Server processes a request and sends a response. Browser interprets the response and may:

- Issue a new request without user knowledge (i.e. follow a 302 response to its location)
- Render the response in the negotiated format

Static Pages vs. Application Processed vs. Forwarded

7.2 Rails command

@TODO

7.2.1 Rails generator

@TODO

7.2.2 Rails engines

@TODO

7.3 Rails application templates

7.4 Effective Use of ActiveRecord

> Active Records are special forms of DTOs [Data Transfer Objects]. They are data structures with public variables; but they typically have navigational methods like *save* and *find*. Typically these Active Records are direct translations from database tables, or other data sources.
>
> Unfortunately we often find that developers try to treat these data structures as though they were objects by putting business rule methods in them. This is awkward because it creates a hybrid data structure and an object.
>
> The solution, of course, is to treat the Active Record as a data structure and to create separate objects that contain the business rules and that hide their internal data (which are probably just instance of the Active Record).
>
> – Martin, Robert C. "Clean Code: A Handbook of Agile Software Craftsmanship (Robert C. Martin Series)"

Traditionally, Rails pushed ActiveRecord objects towards Fat Models. I blame Fat Models, Skinny Controllers. But I am not without blame. I jumped on the Fat Models train.

I remember early in Rails development having MVC related arguements about where things should go. My coworker said "In the controller" and I said "In the model."

It turns out we were both wrong. But we didn't have a noun to attach the behavior to.

We were talking about a Query object; Something that finds the objects that we need. In other cases we needed a Command object; Something that transforms an object.

7.4.1 Single Responsibility Principle

If your class inherits from another, you are signing a contract stating that when the ancestor changes, your class will also change. In other words, your object, begins life with with one reason for changing. And the Single Responsibility Principle says an object should have one reason to change.

7.5 A Non-Exhaustive Curated Collection of Blog-posts

7 Patterns to Refactor Fat ActiveRecord Models Testing from the Outside In

Chapter 8

Testing

> We write tests first because we find that it helps us write better code. Writing a test first forces us to clarify our intentions, and we dont start the next piece of work until we have an unambiguous description of what it should do. — Kent Beck's forward for "Growing Object-Oriented Software, Guided by Tests" by Steve Freeman and Nat Pryce.

8.1 Wisdom is Learning from Other People's Mistakes

My first Rails project started December 2005. I dove in and began writing the production code. I was learning Rails for the first time.[1] It was a volitale period in which Rails and its ecosystem were rapidly evolving.

So much was changing, and I was working at learning it all. I didnt feel I could add "writing tests" to the list of things I could do.

And I learned this was the path of pain. I was writing bugs – every developer does this – and the ecosystem was changing from beneath me. Everything was in motion.

[1]According to David Heinemeier Hansson, "Rails itself is a carefully curated collection of APIs and DSLs.".

My code was sloshing around, collapsing in on itself. And my dependencies were changing, mangling what I had done.[^a_commitment_to_semantic_versioning]

It was a nightmare of fragility. Had I been writing tests, the nightmare would have been real, but there would have been at least one more thing helping me check up on the work I did the day before.

Today, I write tests.

When I am thinking about a solution, I also ask myself "How will I go about verifying the solution?" I capture the answer to the form in my "production" code and the latter in my "test" code. Both are important.

8.2 Types of Tests

As you are exploring testing you may hear about all sorts of tests: Unit, Functional, Integration, Component, System, Exploratory, Feature, Acceptance, etc.

You may hear about Test-Driven Development and Behavior-Driven Development.

There is a lot of discussion regarding tests. At this point, I would recommend thinking about tests as belonging to two categories:

- Isolated
- Integrated

Dr. Alan Kay, an author of Smalltalk, coined the term object-oriented. Dr. Kay posited that objects are similar to biological cells that send each other messages.

Think of isolated tests as testing a single cell. This is where you make sure the cell is operating in a well-structured manner. *Isolated tests make sure I'm writing code the right way.*

Think of integrated tests as tests that coordinate the interaction of multiple cells. This is where you make sure your organism is operating as expected. *Integrated tests make sure I'm writing the right thing.*

I recommend reading The Clean Coder: A Code of Conduct for Professional Programmers by Robert C. Martin. In chapter 8 he explains the various tests, as well as the ratio of each type of test.

8.3 The Documentation Lies

I maintain that the first tattoo I get will be the words “The Documentation Lies.” I have seen well intentioned efforts to write documentation come up short. Effort is spent describing how the system should be working; Or how the system interacts with other systems.

And three months after a documentation push, the documentation is out of date. It provides incorrect instructions. Or does not reflect the current and changing reality.

Part of the cure for lying documentation is to write tests. Tests explain how things are working. They consume the low-level API of your application.

So, write tests that have clear examples of input and output. Draw attention to those tests. Steer your adopters to those tests. Place an effort on making these tests understandable with minimal up front domain knowledge.

For recent projects, Sipity Hydramata::Works and its sibling Hydra Connect Demo, I have made a commitment to grooming a “Getting Your Bearings” section of the README. This section points to tests that document what the system is doing.

And Travis CI asserts the documentation on each commit.

8.4 Writing Tests

Practice writing tests.

If you don’t do it, pick something and practice. Take a look at the Bowling Game Kata or Conway’s Game of Life.

If you write tests, but only after you have written your production code, then try your next problem by first writing the test you would use to verify the behavior. It is a different way of thinking, but it is helpful to consider the numerous collaborators.

8.5 Travis

> Travis CI is a hosted continuous integration service. It is integrated with GitHub and offers first class support for: C, C++, Clojure, Erlang, Go, Groovy, Haskell, Java, JavaScript (with Node.js), Objective-C, Perl, PHP, Python, Ruby, and Scala – http://docs.travis-ci.com/user/getting-started/

If you are writing open source software, please make use of Travis-CI.org. Its free for open-source projects.

8.6 Code Coverage

Software often times makes use of indirection.

It is an active curation process. And as understanding grows and changes, so to does the system. I've learned Rails 0.x, 1.x, 2.x, 3.x, and 4.x. From 2.x to 3.x it was as though I had to learn Rails again.

[^a_commitment_to_semantic_versioning] This was in the day of aparrent arbitrary versioning. Semantic Versioning may have been the goal, but the reality was a hellish landscape. Please take some time to read up on Semantic Versioning and commit to abiding by that discipline.

Chapter 9

Modes of Development

For me, software development is a juggling act. I jump from deep concentration to guiding another developer to answering an unrelated work email over to a chat with a remote developer then off to a meeting.

It is a challenge to remain focused. But for me it is a requirement to get things done.

As I've transitioned into a mentor and lead role, I am also responsible for growing my teams capabilities. This runs contrary to keeping focused.

Over the past year I've worked towards adopting explicit modes of operation. I need to enter into different head spaces.

Look to the venerable and beloved `vi` and `vim`.

It has the concept of modes:

- Normal (Command)
- Insert
- Command-Line
- Replace
- Visual
- Select

VIM Modes Transition Diagram by Marco Fontani

These modes focus on the tasks you are trying to accomplish. Each mode has a single responsibility.

9.1 Its Red, Green, Refactor

Red, Green, Refactor is a hallmark Test Driven Development.

- **Red:** Write a test that will verify something works, before you write the thing to make it work.
- **Green:** Write the thing to make your test pass.
- **Refactor:** Improve on your solution to the problem.

It is a powerful method for solving a problem. But is incomplete.

Before I can get to Red I need to know what I am trying to solve. A proposed solution is a response to an observation. The proposed solution is a hypothetical solution to a real world problem.

Once I have a proposed solution, I forumalate how I will verify my solution. Then I analyze the results of the tests and the solution.

This cleaves close to the scientific method

- Observe
- Conjecture
- Predict
- Test
- Analyze

The Red, Green, Refactor is the manifestation of the Test and Analyze steps of the scientific process.

Important in the Red, Green, Refactor methodology, is my recognition of which mental state in which I am working. There have been too many times where I've been in a Refactor mode and drifted towards writing new features.

What ends up happening is I end up with a very dirty working directory. I have lots of changes and layers of abstraction that are comingled. I end up spending a bit of time using git's interative shell to pick apart the larger ideas.

9.2 Never Unprepared

I am an avid player and facilitator of table top role-playing games. You've been warned.

> Take something complicated, break it up in to manageable parts, and then tackle them over time in order to achieve a goal. – *Phil Vecchione*

In 2012 I picked up a copy of Phil Vecchione's Never Unprepared: The Complete Game Masters Guide to Session Prep. Phil is a gamer at heart, and a project manager by trade. "Never Unprepared" is the intersection of the time management skills of a professional project manager and his hobby.

Phil speaks about the five steps of preparation.

- Brainstorming - Spawn lots of ideas
- Selection - Winnow down the ideas
- Conceptualization - Expand on the selected ideas
- Documentation - Write down meaningful notes for the concepts
- Review - Reconcile the notes for consistency

In identifying the steps, Phil is creating boundaries for the types of tasks he does. In creating these boundaries, he declares the constraints on the time he is working.

There is no place for rejecting ideas during the brainstorming nor documentation phase. Selection is the time for doing so.

More than any software development book, Never Unprepared is a guide for my day to day professional life.

It also helped me think about the different mental modes that I might enter over the course of a work day. What follows is a survey of the various modes that I enter. As a junior developer some of these modes may not be applicable. As a mid-level developer many of them may be applicable.

9.3 Modes that I can think of

What follows is an inventory of the modes of thinking in which I find myself.

9.3.1 Meeting Mode

- **Goal:** Capture group decisions, action items, keep things moving.
- **Method:** Have an agenda. Have a facilitator, note taker, and time keeper. Be quick to appoint small groups.

9.3.2 Thinking on a Solution

- **Goal:** To arrive at a decision that can be acted on.
- **Method:** Go for a walk. Talk it over with someone. Draw and diagram. Prototype code.

9.3.3 Writing an Isolated Test

- **Goal:** To build a supporting case for an integration test.
- **Method:** Use low-overhead collaborators; Mocks and stubs are great, production data structures are also great.

9.3.4 Writing an Integration Test

- **Goal:** To formalize a method for asserting that a solution is acceptable.
- **Method:** Think not on the solution, but how the solution can be verified.

9.3.5 Writing a First Pass Solution

- **Goal:** Get an integration test passing; Assert that something works.
- **Method:** Write a class that attempts to resolve the solution. Look towards collaborators necessary to solve the problem. Keep SOLID principles salient. Jump into isolated tests to flesh out the core class and its contributors.

9.3.6 Refining How a Test Reads

- **Goal:** Create a document that can be used as a reference point into the system
- **Method:** Write the tests for clarity and expressiveness. Convey meaning and how things are assembled. In my experience this focusing on legible and digestable tests is stronger than any README documentation.
- **Caveat:** Documentation lies. Consider documentation that can assert it is correct. That documentation is an automated test.

9.3.7 Documentation

- **Goal:** Convey high level concepts; Why things are being done, not how; The code describes how things are done.
- **Method:** Write brief descriptions. Bullet points are your friend. Write for the web.
- **Caveat:** See Refining How a Test Reads

9.3.8 Writing a Commit Message

- **Goal:** To create a helpful message to future developers
- **Method:** Follow a well established pattern for commit messages; 50 character first line, references to issue tracker incidences, meaningful message body if the concept is complicated.
- **Caveat:** If I am working on a feature that will require multiple commits, I will make "disposable commits" along the way. These commits are mental placeholders. I will revisit them as I prepare a branch for merging.

9.3.9 Preparing a Pull Request

- **Goal:** To present an atomic chunk of work for final review.
- **Method:** Squash commits into logical chunks. Expand on any terse commit messages. Take time to communicate the intentions of the code.

9.3.10 Reviewing a Pull Request

- **Goal:** To make sure the code is doing the right thing (in the right way).
- **Method:** Reconcile the commit message, code comments, code, and associated tickets/stories. Ask questions of the developer.

9.3.11 Word Smithing

- **Goal:** To write meaningful copy for patrons of the software I'm writing.
- **Method:** Stop and think about how the patron will encounter the copy. What needs to be said?

9.3.12 Exploring an External Dependency

- **Goal:** To assess and determine the viability of an external solution.
- **Method:** Read the README. Check the test suite for existence, legibility, and clarity. Check other adopters (i.e. ruby-toolbox.com). Attempt to run the tests. Check code coverage and quality (i.e. coveralls and code climate). Check commit messages for clarity. Look at frequency of contributions. Look for similar solutions.

9.3.13 Writing Blog Posts

- **Goal:** To assimilate lessons learned and convey those lessons in a digestible format.
- **Method:** Write concise posts. Build from previous posts. Explore ideas. Offer links to supporting evidence. Highlight lessons learned. Take a risk. Accept amendments.

9.3.14 Refactoring

- **Goal:** To move the code to a more expressive state.
- **Method:** Create a new branch. Baby step commits, the messages are disposable. Craft the final commit message. If new ideas bubble up, create another branch and explore. A flurry of commits, sometimes spanning multiple branches. Each branch an atomic concept.
- **Caveat:** Once the refactor is complete, prepare a pull request.

9.3.15 Reviewing a branch not yet ready for pull request review

- **Goal:** To help give shape to the solution as it is worked on by others

- **Method:** Ask lots of questions. Why this particular name? What do you mean here? Is there an abstraction you are wrestling with?

9.3.16 Story Triage

- **Goal:** To expand user stories to more actionable developer tasks
- **Method:** Review the existing story and think about the impact on the existing system. Think about strange states. Ask about all the UI components; What should the copy be? What should the UI experience be? Think of other related things that could be abstracted.

9.3.17 Unearth a Problem

- **Goal:** To identify a problem and how this problem manifests
- **Method:** Ask lots of questions. Poke and prod at assumptions. Review commit logs to see if there is churn on a particular constellation of files. Challenge each and every “We should do [better at]…” there is a problem in there, noodle on it. Roll it around.

Chapter 10

RSpec

10.1 Documentation Format

10.2 rspec-given

@TODO

10.3 Custom matchers

10.3.1 rspec-yenta

I like writing custom Rspec matchers. Other people write custom Rspec matchers (e.g. rspec-rails and rspec-html-matchers).

A coworker of mine asked me how to find all of the matchers for one of our projects. My answer was to create `rspec-yenta`.

```
$ rake -T yenta

rake yenta   # Find all of your rspec matchers and output them to STDOUT
```

Configure `rspec-yenta` for your project by following `rspec-yenta`'s README. Then run `rake yenta`.

Below is a portion of output — albeit modified — from one of my projects.

```
$ rake yenta

be            ./path/to/rspec-expectations/lib/rspec/matchers.rb:221
be_a          ./path/to/rspec-expectations/lib/rspec/matchers.rb:227
be_a_kind_of  ./path/to/rspec-expectations/lib/rspec/matchers.rb:253
be_a_new      ./path/to/rspec-rails/lib/rspec/rails/matchers/be_a_new.rb:73
```

Chapter 11

Acceptance Testing

I have maintained that a software developer is analogous to an author. And my production application is the book I am writing. Few people would consider publishing a book without first having an editor.

In 2014, I began working with my first editor. His name is Chris. He is our Lead Quality Assurance Developer.

His job is to try to break the systems that I make. And I am thankful for his dedication to the task.

He helps me drag features across the “mostly done” line and into the coveted “done done” area. He bring closure to a task. Ensuring that the nuances of a feature is fleshed out and not a half-baked solution.

We have had several discussions about our acceptance test tooling. What follows is our attempt to distill those conversations.

11.1 Understand your audience for the acceptance tests

Who will be reading these tests?

- Non-technical peole
- Technical-minded people

- Programmers

What is the purpose and scope of the software?

- Command line tools
- Web-based APIs
- Web-based UI in all its glorious CSS and Javascript
- An application
- A framework for building an application

Who will assist in writing the acceptance tests?

- Programmers
- Project Managers
- Patrons

How much effort will you place on automating the acceptance tests?

- Minimal effort, as we will click around the system
- Infinite effort, automate all the things
- Somewhere in between, most tasks will be automated yet we acknowledge a precision in language is required

11.2 Address the Ecosystem

For the past two years I've worked under the following pattern:

- A Rails::Engine for defining the abstract application, and teasing apart isolated concerns.
- A Rails application that leverages the Rails::Engine, providing configuration and modifications.

It is much more of an art than science, but what I've found is the following:

Gems and Rails::Engines have an audience of programmers. Make sure the acceptance tests are meaningful and expressive to them. I prefer to write well-considered RSpec scenarios[1] or Capybara scenarios if I have a UI.

The Rails applications that I work on have a varied audience. Asking them to read RSpec scenarios or even Capybara scenarios is too much. Instead we are pushing to use Fitnesse as our acceptance test methodology.

Fitnesse has one killer feature. Anyone can write the documentation of how the system "is doing things."[2] And the documentation can be written such that it is verified by test code.

11.3 Methodology

For acceptance tests, focus on writing clear examples; A tight focus on:

- the input parameters
- the expected output
- the *one* function call that turns input into output

[1]I contribute to ProjectHydra, and prior to my participation, it was agreed that ProjectHydra would use RSpec. It can be very powerful for writing expressive tests, with a focus on "natural" language. But in its accomodation for natural language, RSpec provides many different ways of saying the same thing. In other words, it can be very confusing to get started.

[2]Remember, the documentation lies

11.4 Tools of the Trade

11.4.1 Capybara

> Capybara helps you test web applications by simulating how a real user would interact with your app. It is agnostic about the driver running your tests and comes with Rack::Test and Selenium support built in. WebKit is supported through an external gem. – https://github.com/jnicklas/capybara

This is my goto tool for testing UI features. It uses a familiar syntax - ruby code - to express the tests.

If you find yourself in a position with lots of Capybara tests, take some time to craft Page Objects to help keep your Capybara CSS selectors tame. SitePrism is a gem that is an implementaiton of a Page Object Model.

When you execute Capybara tests, you can switch your driver to show the browser interaction. Watch as Capybara fills out form fields, clicks buttons, and asserts an expectation. Now imagine doing that as part of your demo; You may need to add some sleep/pause logic.

11.4.2 Cucumber

> Cucumber is a tool for running automated tests written in plain language. Because they're written in plain language, they can be read by anyone on your team. Because they can be read by anyone, you can use them to help improve communication, collaboration and trust on your team. – https://github.com/cucumber/cucumber/

Read the quote again. Ask yourself?

- Who will be writing the plain language?
- Who will be writing the code that assert the plain language?

In my experience, Cucumber requires a precision in writing that creates a lot of overhead.

Writing Cucumber specs requires a commitment from the team.

- Who will be reading these tests?
- Who will be helping craft the scenarios?
- Will team members take the time to read these things?

Using Cucumber can be a powerful tool of collaboration, but all collaboration comes with a time cost. Make sure it is worth it.

Ask your team:

- What kind of documentation they want to see?
- Who the documentation is for?

Get a sense of the audience for the tests, because you will want to craft your examples to reflect their needs.

11.4.3 Fitnesse

As of this writing, I have yet to make use of Fitnesse.

It looks to be useful in growing both documentatino and public facing assertions that things are working.

Chapter 12

Miscellany

Aside from Google and Stack Exchange, here are a few resources that rise to the top.

12.1 Ruby Rogues

Ruby Rogues is a free podcast with panelists speaking on all things Ruby and more. It is well produced and I find it very informative.

12.2 Ruby Tapas

Ruby Tapas is a bi-weekly screencast of short forays into intermediate and advanced Ruby topics.

12.3 Railscasts

An aging collection of Ruby on Rails screencasts. There is a lot of information that still holds value. Take a look at the Free Episodes

12.4 Thoughtbot's Blog

A diverse collection of blog posts by a preeminent Rails development shop.