

gem that

by James Adam

hello, I'm James Adam, and I'm going to talk a little bit about building gems.

the title for my talk is 'gem that', which makes me think the opening slide should be more like



gem that!

by James Adam!

wow! Gem that! but realistically it should be more like

(a constructive rant, and then gem-this)

(and writing gem commands)

(by James Adam)

a constructive rant, and then what I did – i.e. gem–this, and then some information about writing gem commands.

I am going to have to drive at around 88 MPH to get through this, so it's probably best to save questions for the end, otherwise I might crash.

but first

building gems - a primer

it's worth spending a few seconds considering the background of building gems.

in it's simplest form, building a gem is simply a case of

my_gem.gemspec

```
Gem::Specification.new do |s|  
  s.name = "my_gem"  
  s.version = "1"  
  
  s.authors = ["James Adam"]  
  s.date = "2009-12-10"  
  s.description = "What it does"  
  s.email = "james@lazyatom.com"  
  s.files = ["Rakefile", "lib/thing.rb"]  
  s.homepage = "http://lazyatom.com"  
  s.require_paths = ["lib"]  
  
  # etc  
end
```

constructing a 'gemspec', which is ruby code that describes a gem, what it does, what files it should include and which dependencies it has, and then

```
$ gem build my_gem.gemspec
Successfully built RubyGem
Name: my_gem
Version: 1
File: my_gem-1.gem
$ ls *.gem
my_gem-1.gem
$
```

building that gem using the **'gem build'** command.

it's really not that **sophisticated**, and the capabilities are **built right into rubygems itself**.

however, given some of the tools, you might not realise that it is so simple.

my motivation

secondly, I should explain **my own perspective** here. As a developer, I quite often toy around with an idea before deciding that it might be suitable for other people to use.

Often I've already put a bit of effort into **organising the code**, writing some rake tasks and so on.

So when it comes to turning it into a gem, I am looking for a tool that is going to support **me**.

So what should I use?

hoe

the 'godfather' of gem creation

well, the original daddy of gem creation was 'hoe', written by Ryan Davis. It provides a command to run when you're setting up your project – 'sow'.

So let's see what happens when we run that.


```
$ sow hoe_project
cp -r /Users/james/.rvm/gems/ree/1.8.6%rubymanor_gem_that_talk/gems/hoe-2.3.3/
template /Users
james/.hoe_template
chmod 644 /Users/james/.hoe_template/bin/file_name.erb /Users/james/.hoe_template/
History.txt.erb /Users/james/.hoe_template/lib/file_name.rb.erb /Users/
james/.hoe_template/Manifest.txt.erb /Users/james/.hoe_template/Rakefile.erb /Users/
james/.hoe_template/README.txt.erb /Users/james/.hoe_template/test/
test_file_name.rb.erb /Users/james/.hoe_template/.autotest.erb
chmod 755 /Users/james/.hoe_template/bin/file_name.erb
cp -r /Users/james/.hoe_template hoe_project
erb: .autotest.erb
erb: History.txt.erb
erb: Manifest.txt.erb
erb: README.txt.erb
erb: Rakefile.erb
erb: bin/file_name.erb
erb: lib/file_name.rb.erb
erb: test/test_file_name.rb.erb
mv .autotest.erb .autotest
mv History.txt.erb History.txt
mv Manifest.txt.erb Manifest.txt
mv README.txt.erb README.txt
mv Rakefile.erb Rakefile
mv bin/file_name.erb bin/hoe_project
mv lib/file_name.rb.erb lib/hoe_project.rb
mv test/test_file_name.rb.erb test/test_hoe_project.rb
(... cont ...)
```

Wow. I mean... firstly, look at all that stuff. Where do I start? It's created an autotest file, and a Manifest, and a README, and it's still not even finished

```
$ sow hoe_project  
(.... cont ....)
```

... done, now go fix all occurrences of 'FIX':

```
hoe_project/Rakefile:7: # developer('FIX', 'FIX@example.com')  
hoe_project/README.txt:3:* FIX (url)  
hoe_project/README.txt:7:FIX (describe your package)  
hoe_project/README.txt:11:* FIX (list of features or problems)  
hoe_project/README.txt:15: FIX (code sample of usage)  
hoe_project/README.txt:19:* FIX (list of requirements)  
hoe_project/README.txt:23:* FIX (sudo gem install, anything else)  
hoe_project/README.txt:29:Copyright (c) 2009 FIX
```

```
$
```

now I am starting this project with a ton of stuff that needs to be **fixed**; I am starting my project with a bunch of debt that I need to pay off!

And what is this Manifest? do I have to use autotest? Well, OK, let's take a look at the Rakefile.

hoe Rakefile

```
# -*- ruby -*-  
require 'rubygems'  
require 'hoe'  
  
Hoe.spec 'hoe_project' do  
  # developer('FIX', 'FIX@example.com')  
  
  # self.rubyforge_name = 'hoe_projectx'  
  # if different than 'hoe_project'  
end  
  
# vim: syntax=ruby
```

What I really dislike about Hoe's Rakefile is that it requires me to learn all about Hoe before I can really get started. All it's given me is a hint about where I should put my name and email.

hoe Rakefile

```
Hoe .spec 'hoe_project' do  
  # ???  
end
```

What is supposed to go here? **Pour through the RDoc**, when I'd rather be writing my own code.

So sure, there's a learning curve involved when using Hoe, but the really major turnoff for me is that hoe is a virus...

hoe
is a
fucking
virus



What I mean by that when you install a gem that was created with Hoe, it will also install Hoe itself.

Hoe is controlling the generation of the gem, and Hoe adds itself as a dependency.

That immediately pisses me off.
Now I should qualify this by saying that

hoe
was a
fucking
virus



Hoe was a virus, because in recent times, it has relented, and doesn't require itself to be installed by every gem that uses it.

but **IT'S TOO LATE HOE. YOU ARE DEAD TO ME.**

So what else can I try?

newgem

- Dr Nic

NewGem, from Dr Nic Williams

Oh, which means incidentally that this presentation should be called



gem that!

by Dr James Adam!

Gem That!

by DOCTOR James Adam, but anyway, there's

newgem

- generating structure, like the rails command
- extensible with other ‘generators’
- pretty bat-shit mental.

NewGem, from Dr Nic Williams (<http://drnicwilliams.com/2006/10/11/generating-new-gems/>)

which seems to be a **product of the post-rails age**, and fully embraces the ‘generative’ style of programming.

Lets see what happens when we try to use that

```
$ newgem newgem_project
  create
  create lib/newgem_project
  create script
  create History.txt
  create Rakefile
  create README.rdoc
  create PostInstall.txt
  create lib/newgem_project.rb
dependency install_test_unit
  create test
  create test/test_helper.rb
  create test/test_newgem_project.rb
dependency install_rubigen_scripts
  exists script
  create script/generate
  create script/destroy
  create script/console
  create Manifest.txt
  readme readme
```

Important

=====

- * Open Rakefile
- * Update missing details (gem description, dependent gems, etc.)

\$

Look at all that stuff – PostInstall? script/console? script/generate?

It's obvious that newgem is **trying to be comprehensive** in the sorts of libraries that it might be used with, but this does seem a bit like overkill.

And look at all the Rake tasks it generates.



```
$ rake -T
rake announce      # publish # Announce your release.
rake audit         # test   # Run ZenTest against the package.
rake check_extra_deps # deps  # Install missing dependencies.
rake check_manifest # debug # Verify the manifest.
rake clean        #       # Clean up all the extras.
rake clobber_docs # publish # Remove rdoc products
rake clobber_package # package # Remove package products
rake config_hoe   # debug # Create a fresh ~/.hoerc file.
rake debug_email  # publish # Generate email announcement file.
rake debug_gem    # debug # Show information about the gem.
rake default      # test  # Run the default task(s).
rake deps:email   # deps  # Print a contact list for gems dependent on this gem
rake deps:fetch   # deps  # Fetch all the dependent gems of this gem into tarballs
rake deps:list    # deps  # List all the dependent gems of this gem
rake docs         # publish # Build the RDOC HTML Files
rake gem          # package # Build the gem file newgem_project-0.0.1.gem
rake gemspec      # newgem # Generate a newgem_project.gemspec file
rake generate_key # signing # Generate a key for signing your gems.
rake install_gem  # package # Install the package as a gem.
rake install_gem_no_doc # newgem # Install the package as a gem, without generating documentation(ri/rdoc)
rake manifest     # manifest # Recreate Manifest.txt to include ALL files to be deployed
rake multi        # test  # Run the test suite using multiruby.
rake package     # package # Build all the packages
rake post_blog   # publish # Post announcement to blog.
rake post_news   # publish # Post announcement to rubyforge.
rake publish_docs # publish # Publish RDoc to RubyForge.
rake redocs      # publish # Force a rebuild of the RDOC files
rake release     # package # Package and upload the release.
rake release_sanity # package # Sanity checks for release
rake release_to_rubyforge # package # Release to rubyforge.
rake repackage   # package # Force a rebuild of the package files
rake ridocs      # publish # Generate ri locally for testing.
rake test        # test  # Run the test suite.
rake test_deps   # test  # Show which test files fail when run alone.
$
```

By my count, that's THIRTY FOUR rake tasks, including posting to a blog, installing the gem locally but WITHOUT any rdoc, and **printing a list of the email addresses of the authors for the gems that THIS gem depends on...**

I mean... I am pretty sure I don't need that.

We're still not done. That was the simple mode – and we can provide it with additional 'generators'. Lets try it with a few more



```
$ newgem -i cucumber -i website newgem_ultra_project -i shoulda
  create
  create lib/newgem_ultra_project
  create script
  create History.txt
  create Rakefile
  create README.rdoc
  create PostInstall.txt
  create lib/newgem_ultra_project.rb
dependency install_test_unit
  create test
  create test/test_helper.rb
  create test/test_newgem_ultra_project.rb
dependency install_cucumber
  create features/step_definitions
  create features/support
  create features/development.feature
  create features/step_definitions/common_steps.rb
  create features/support/env.rb
  create features/support/common.rb
  create features/support/matchers.rb
dependency install_website
  create website/javascripts
  create website/stylesheets
  create config
  exists script
  create website/index.txt
  create website/index.html
  create config/website.yml.sample
  create script/txt2html
dependency plain_theme
  exists website/javascripts
  exists website/stylesheets
  create website/template.html.erb
  create website/stylesheets/screen.css
  create website/javascripts/rounded_corners_lite.inc.js
dependency install_shoulda
  exists test
  create tasks
  force test/test_newgem_ultra_project.rb
  force test/test_helper.rb
  create tasks/shoulda.rake
dependency install_rubigen_scripts
  exists script
  create script/generate
  create script/destroy
  create script/console
  create Manifest.txt
  readme readme

Important
=====

* Open Rakefile
* Update missing details (gem description, dependent gems, etc.)
$
```

Now newgem is creating me a website, and a bunch of stub tests and cucumber features for code that I haven't written yet! Thanks, newgem!

I am kind-of speechless, because faced with this all of a sudden my little idea for a useful library seems **unworthy** of being a gem, because it surely doesn't warrant all of this support behind its existence.

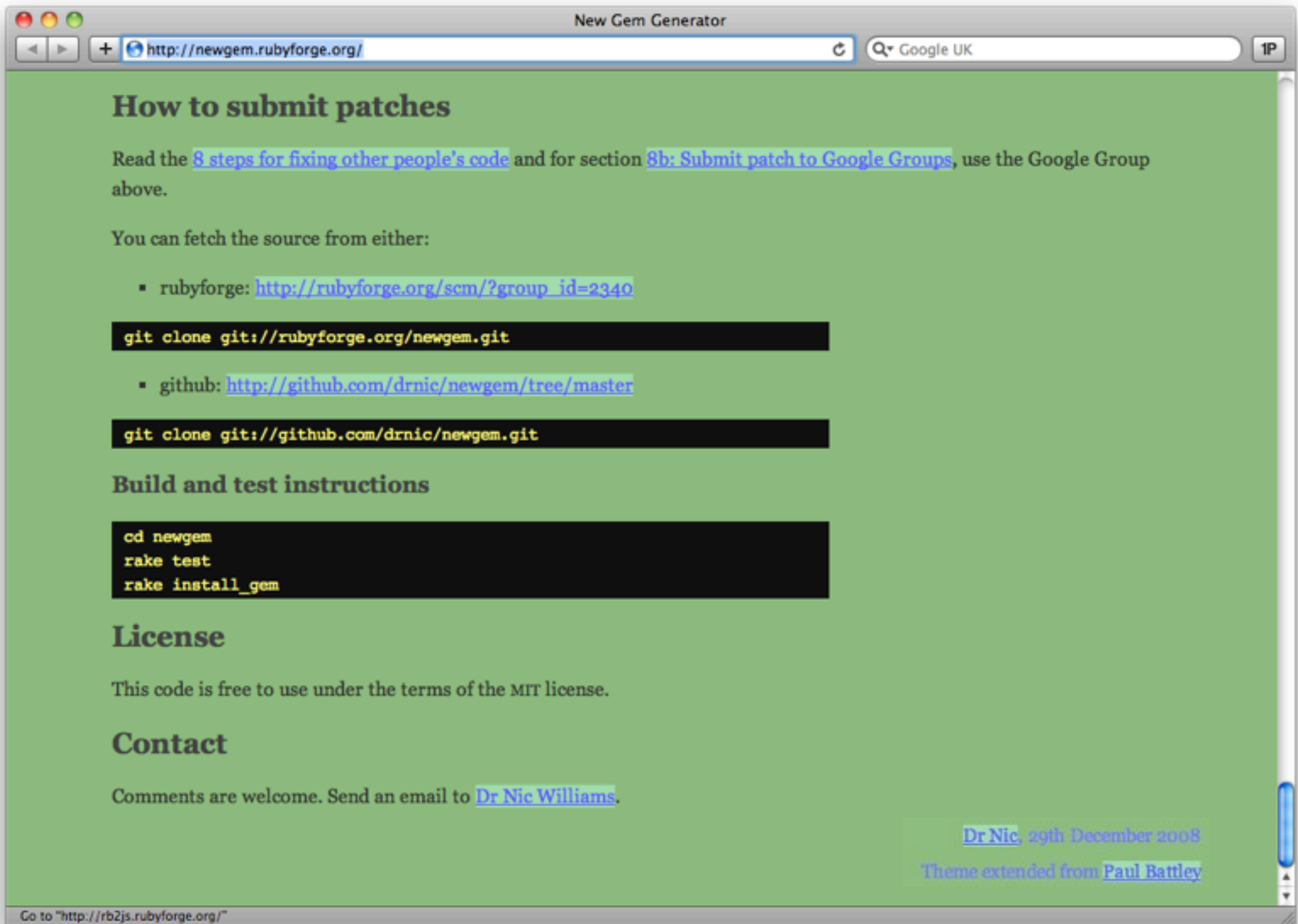
I started to question my grip on reality at this point

```
$ rake -T sanity  
rake release_sanity # ...
```

WHaTEVeR
U SaY
Dr NiC!

and found that newgem has a rake task for that too. Doctor Nic thinks of everything!

The one genuinely good thing about newgem, however, is that every website it creates



has a link to Paul Battley on it, increasing his pagerank.

Nice work, Paul! It's a shame that you had to make a deal with the devil to do it.

So what now?



[Dr Nic](#), 29th December 2008

Theme extended from [Paul Battley](#)

has a link to Paul Battley on it, increasing his pagerank.

Nice work, Paul! It's a shame that you had to make a deal with the devil to do it.

So what now?

echoe

= (hoe) - (it being a dependency)

echoe is a fork of 'hoe' by evan weaver, whose main purpose was to avoid having itself as a dependency.

It doesn't provide a generator, with is a positive. Instead you have to add a few lines to your Rakefile

echoe Rakefile

```
require 'echoe'  
Echoe.new('gem_name')
```

Here's the contents of an echoe Rakefile

But it's still very opaque. How do I add new stuff? How do I change the description?

Well, you do it a bit like this

echoe Rakefile

```
Echoe.new("vanilla.rb") do |p|  
  p.author = "James Adam"  
  p.summary = "A talk about this would've  
              been awesome"  
  p.url = "http://interblah.net"  
  p.runtime_dependencies = ["soup >=1.9.9"]  
  # etc... ?  
end
```

But to figure **that** out you still need to **dive into the RDoc**, and even then it's not simple.

For example, the **only way to change the version** of a gem is to add a line to the **CHANGELOG** file.

And even though echoe doesn't declare itself as an **explicit dependency** of the gems it creates, if you want to put other tasks in the Rakefile that people can run **without** installing echoe, you need to

echoe Rakefile

```
begin
  require 'echoe'
  Echoe.new('my_gem')
rescue LoadError
  # probably nothing
end
# the rest of your Rakefile
```

wrap the echoe code in a block to capture the exception! Ugly.

Next.

gemhub

(somewhat of a mystery to me)

So when I proposing this talk on the mailing list, someone mentioned gemhub (<http://github.com/dcrec1/gemhub>) by **Diego Carrion**, who claims on his blog to '**love bits, tits and beers**', but we'll try not to hold that against him.

If we run it's generator

```
$ gemhub gemhub_project
```

```
  create
```

```
  create  lib
```

```
  create  spec
```

```
  create  lib/gemhub_project.rb
```

```
  create  spec/gemhub_project_spec.rb
```

```
  create  README.textile
```

```
  create  Rakefile
```

```
$
```

we see something a bit more sane, but at this point I'm so agitated that

- * when I notice it's telling me that I should be using **RSPEC** for my tests when I prefer Shoulda,

- * and **TEXTILE** for my documentation when I think that Markdown is clearer for plaintext,

... I start weeping and saying "No, No gemhub, you could've been right for me, but we're TOO DIFFERENT." Next.



jeweler

the choice of a new generation

So then we have Jeweler (<http://github.com/technicalpickles/jeweler>), which I would say is 'top dog' among the newest crop of Ruby developers; lets see what it does to offend me, yes?

```
$ jeweler jeweler_project
  create .gitignore
  create Rakefile
  create LICENSE
  create README.rdoc
  create .document
  create lib
  create lib/jeweler_project.rb
  create test
  create test/helper.rb
  create test/test_jeweler_project.rb
Jeweler has prepared your gem in jeweler_project
$
```

OK, this is not too bad, certainly not as bat-shit mental as newgem (although I don't understand what .document is).

If we look at the Rakefile

jeweler Rakefile

```
begin
  require 'jeweler'
  Jeweler::Tasks.new do |gem|
    gem.name = "jeweler_project"
    gem.summary = %Q{summary of your gem}
    gem.email = "james@lazyatom.com"
    gem.homepage = "http://github.com/..."
    gem.authors = ["James Adam"]
    # etc ...
  end
  Jeweler::GemcutterTasks.new
rescue LoadError
  puts "Jeweler not available...."
end
```

ok, that's not too bad, and even though it's wrapped stuff in the rescue block

It's also wrapping the gemspec in this Jeweler::Tasks thing, the 'gem' variable that it's yielding *is* an actual gemspec so you have complete control over what is going on there.

But then we take a look at the rake tasks it has added


```
$ rake -T
rake check_dependencies
rake check_dependencies:development
rake check_dependencies:runtime
rake gemcutter:release
rake gemspec:debug
...
rake git:release
rake github:release
rake install
...
rake release
...
rake version:bump:major
rake version:bump:minor
rake version:bump:patch
rake version:write
$
```

and there's all this cruft about dependencies, and github releases, and version bumping, that I JUST. DON'T. NEED.

jeweler

- zeitgeisty
- tailored for github, apparently
- still provides tasks for bumping versions, publishing
- still provides a generator

it's just way too much. My focus has shifted from making my little bit of code available, to servicing the needs of this overly-capable swiss-army-knife tool.

And it occurs to me at this point that really what's going is

too opinionated

- don't need a tool to manage versions, websites, release notes, etc
- don't need a generator
- why so opaque?

these tools are too opinionated. They're not for building gems, they're all for **managing your development workflow**, from start to finish – from the first spark of the idea, to the blog post announcing your latest release.

And that's too much for me. What I need is

what I want

- turn existing code into a gem

- *the fuck* get out of my way

a tool that will turn some existing code I have into a gem

and then get the fuck out of my way.

And oh yeah, have you seen what happens if we try and apply any of these 'gem' tools to some code that you've already written?

```
$ sow my_code
```

```
Project my_code seems to exist  
$
```

Here's hoe (or sow – yes, it's confusing), discovering that my code 'already seems to exist', as if it cannot contemplate such a thing was possible.

```
$ jeweler my_code
```

The directory my_code already exists. Maybe move it out of the way before continuing?

```
$
```

And jeweler, suggesting that ‘maybe I should move it out of the way’.
Maybe jeweler shouldn’t be so bloody passive aggressive!

The only tool that comes even close is gemhub

```
$ gemhub my_code
  exists
  create lib
  create spec
  create lib/my_code.rb
  create spec/my_code_spec.rb
  create README.textile
overwrite Rakefile? (enter "h" for help) [Ynaiqd] h
Y - yes, overwrite
n - no, do not overwrite
a - all, overwrite this and all others
i - ignore, skip any conflicts
q - quit, abort
d - diff, show the differences between the old and the new
h - help, show this help
overwrite Rakefile? (enter "h" for help) [Ynaiqd]
```

but when it notices that I already have a Rakefile, it offers to either overwrite it, or not, or handily show me how wildly different my ideas for the Rakefile were compared to its own, for me to presumably memorise and retype later on?

sigh.

GEM THIS

This is why I wrote 'gem-this'.

gem this

- produces a simple Rakefile
- builds your gem
- maybe does your docs
- release to rubyforge

it only deals with one file – the Rakefile – and as a result, it only has one dependency: Rake

The Rakefile it creates only does three things – builds gems, builds docs, and releases to rubyforge.

And in fact, now that gemcutter is the defacto host

gem this

- produces a simple Rakefile
- builds your gem
- maybe does your docs
- ~~release to rubyforge~~

It's unlikely that I'll keep that code in.

So how do you use it? There's no generator, since it presumes you are smart enough to have already set things up

```
/tmp $ mkdir new_thing
```

```
/tmp $ cd new_thing
```

```
new_thing $ gem-this
```

```
Writing new Rakefile
```

```
$
```

So once you've created your project, and hacked away at it a bit, doing whatever (the fuck) you want, you just run the 'gem-this' command in the directory, and it will create a Rakefile for you.

gem this Rakefile

```
# This builds the actual gem. For details of what  
all these options mean, and other ones you can  
add, check the documentation here:
```

```
#
```

```
# http://rubygems.org/read/chapter/20
```

```
#
```

```
spec = Gem::Specification.new do |s|
```

```
  # Change these as appropriate
```

```
  s.name           = "existing_project"
```

```
  s.version        = "0.1.0"
```

```
  s.summary        = "What this thing does"
```

```
  s.author         = "James Adam"
```

```
  # etc...
```

At the top of this Rakefile is the raw Gem Spec, including comments explaining what each bit is for, and most importantly a link to more documentation so you can really figure things out.

gem this Rakefile

```
# This task actually builds the gem.
Rake::GemPackageTask.new(spec) do |pkg|
  pkg.gem_spec = spec
end

# Generate documentation
Rake::RDocTask.new do |rd|
  rd.rdoc_files.include("lib/**/*.*rb")
  rd.rdoc_dir = "rdoc"
end
```

Underneath is the actual task to create the gem, and the RDoc.

If you have a **bin** directory, it will hook that up. If you have a **test** or a **spec** directory, it will create tasks and dependencies in the case of **rspec**. If you've already got the project in **git**, it will ignore some files for you.

But what if you've already got a Rakefile with some rake tasks that you've been using while you were hacking?

```
/tmp $ cd old_thing  
old_thing $ gem-this  
Appending to existing  
Rakefile  
old_thing $
```

so if you run ‘gem-this’ in a directory that already has a Rakefile, it will simply append everything it generates to the end of your file.

You can then move things around and edit them as you see fit.

Simple.

```
$ gem-this
```

```
• • •
```

```
$ gem this
```

```
• • •
```

```
$ gem push
```

```
• • •
```

```
$ gem open
```

gem install open_gem



so I've been typing 'gem-this', but we can actually type 'gem this', as if the 'this' command was a part of rubygems.

This is a gem command, and it's how 'gem push' works for gemcutter, and 'gem open' works too (that's a handy tool by the way, check it out).

gem commands

These are examples of 'gem commands'.

It's really quite simple to add commands to rubygems.

the secret sauce

- subclass `Gem::Command` ...
- ... in a file called `rubygems_plugin.rb` ...
- ... and ensure it's in the `$LOAD_PATH`

All you need to do it create a subclass

... and ensure it's in the `LOAD_PATH` of your gem (typically that's the lib directory)

... and rubygems will find it.

my_gem/lib/rubygems_plugin.rb

```
require 'rubygems/command_manager'  
require 'rubygems/command'  
  
class TestCommand < Gem::Command  
  def initialize  
  end  
  
  def summary  
  end  
  
  def execute  
  end  
end
```

In the subclass we need to implement these three methods: initialize, summary and execute.

(As far as I can tell the name of the subclass needs to be in the form of <your name> Command, so i)

my_gem/lib/rubygems_plugin.rb

```
require 'rubygems/command_manager'  
require 'rubygems/command'
```

```
class TestCommand < Gem::Command  
  def initialize  
  end  
  
  def summary  
  end  
  
  def execute  
  end  
end
```

In the subclass we need to implement these three methods: initialize, summary and execute.

(As far as I can tell the name of the subclass needs to be in the form of <your name> Command, so i)

my_gem/lib/rubygems_plugin.rb

```
class TestCommand < Gem::Command
  def initialize
    super 'name', 'short description', {:debug => false}
    add_option('-d', '--debug', 'guess!') do |d, options|
      options[:debug] = d
    end
  end

  # def summary
  # def execute
end
```

in the initialize method, you should call super with the name of your command, a short description, and any default options.

You can then add additional options that will be picked up from the command line, in a way similar to opt-parse.

my_gem/lib/rubygems_plugin.rb

```
class TestCommand < Gem::Command
  def initialize
    super 'name', 'short description', {:debug => false}
    add_option('-d', '--debug', 'guess!') do |d, options|
      options[:debug] = d
    end
  end

  # def summary
  # def execute
end
```

in the initialize method, you should call super with the name of your command, a short description, and any default options.

You can then add additional options that will be picked up from the command line, in a way similar to opt-parse.

my_gem/lib/rubygems_plugin.rb

```
class TestCommand < Gem::Command
  def initialize
    super 'name', 'short description', {:debug => false}
    add_option('-d', '--debug', 'guess!') do |d, options|
      options[:debug] = d
    end
  end

  # def summary
  # def execute
end
```

in the initialize method, you should call super with the name of your command, a short description, and any default options.

You can then add additional options that will be picked up from the command line, in a way similar to opt-parse.

my_gem/lib/rubygems_plugin.rb

```
class TestCommand < Gem::Command
  # def initialize

  def summary
    "What this command does..."
  end

  # def execute
end
```

the summary command is fairly self-explanatory – this just needs to be a string

my_gem/lib/rubygems_plugin.rb

```
class TestCommand < Gem::Command
  # def initialize

  # def summary

  def execute
    puts "You ran me with
        #{options.inspect}"
  end
end
```

and finally the execute command is where you kick off whatever your command actually does. In this method you have access to the options hash, which includes the arguments passed on the command line and any option flags that you've defined.

The last thing you need to do is register the command with Rubygems

my_gem/lib/rubygems_plugin.rb

```
class TestCommand < Gem::Command
  # def initialize

  # def summary

  def execute
    puts "You ran me with
        #{options.inspect}"
  end
end
```

and finally the execute command is where you kick off whatever your command actually does. In this method you have access to the options hash, which includes the arguments passed on the command line and any option flags that you've defined.

The last thing you need to do is register the command with Rubygems

my_gem/lib/rubygems_plugin.rb

```
# the command class
```

```
# ...
```

```
Gem::CommandManager.instance.  
  register_command :test
```

This is simply a case of adding this line at the bottom of the `rubygems_plugin` file.

As an example, here's the `gem` command for `gem-this`

gem_this/lib/rubygems_plugin.rb

```
%w(rubygems/command_manager rubygems/command
gem_this).each { |lib| require lib }

class ThisCommand < Gem::Command
  def initialize
    super 'this', GemThis::SUMMARY, :debug => false
    add_option('-d', '--debug', GemThis::DEBUG_MESSAGE) do |d, options|
      options[:debug] = d
    end
  end

  def summary; GemThis::SUMMARY; end

  def execute
    GemThis.new(File.basename(Dir.pwd), options[:debug]).create_rakefile
  end
end

Gem::CommandManager.instance.register_command :this
```

Most of the logic is delegated into the 'GemThis' class, so it can be shared with the actual gem-this bin

**FUCK
YOUR
CONVENTIONS**

I promised someone I wrote include this slide, so here it is.

github.com/lazyatom/gem-this

```
gem install gem-this
```

Here's the github URL – thanks for listening!

james@lazyatom.com

github.com/lazyatom/gem-this

```
gem install gem-this
```

<http://gofreerange.com>

Here's the github URL – thanks for listening!