What is Hoe?

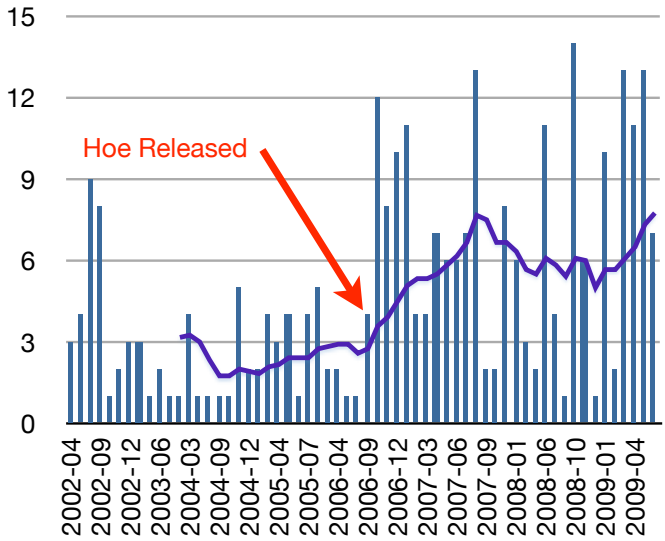
Hoe is a library that provides extensions to rake to automate every step of the development process from genesis to release. It provides project creation, configuration, and a multitude of tasks including project maintenance, testing, analysis, and release. We found rake to be an incredible vehicle for functionality _in the abstract_, but decidedly lacking in concrete functionality. We filled in all the blanks we could through a "hoe-spec":

```
require "hoe"
Hoe.spec "project_name" do
  developer "Ryan Davis", "ryand-ruby@zenspider.com"
  # ...
end
```

This hoe-spec specifies everything about your task that is different from the defaults and from that, creates a multitude of tasks and a gemspec used for packaging and release. When you update hoe, you update all your projects that use hoe. That's it. Nothing more is needed. Everything is DRY (Don't Repeat Yourself).

A Brief History of Hoe

Hoe was extracted from pain, not from fun. It was decidedly ****not**** written in a vacuum. Pain to me is repetition and mindless/needless work. At the time of this writing, a subset of seattle.rb has done 344 releases across 46 products and all but 86 of those releases were done with Hoe. An inordinate amount of time and effort was put into keeping them in sync with each other. In other words, we know what we're talking about here and it ain't pretty.



Every time we found a new task that was useful to one project, it was probably useful to the rest of them... but used in a _slightly_ different way. Resolving those edits across all the projects took time away from writing fun/good/useful code. Every time we found a snafu in our release process and wanted to improve it, we had to propagate those changes lest we have another snafu. In short, we had code duplication across our projects, but on the release/package/process side. We weren't DRY and at the time, there wasn't much available for process-oriented libraries. Through this pain, Hoe was born.

Why Use Hoe?

Deployment, the DRY way

Hoe focuses on keeping everything in its place in a useful form and intelligently extracting what it needs. As a result, there are no extra YAML files, config directories, ruby files, or any other artifacts in your release that you wouldn't already have.

README.txt

Most projects have a readme file of some kind that describes the project. Hoe is no different. The readme file points the reader towards all the information they need to know to get started including a description, relevant urls, code synopsis, license, etc. Hoe knows how to read a basic rdoc formatted file to pull out the description (and summary by extension), urls, and extra paragraphs of info you may want to provide in news/blog posts.

History.txt

Every project should have a document describing changes over time. Hoe can read this file (also in rdoc) and include the latest changes in your announcements.

Manifest.txt

Every project should know what it is shipping. This is done via an explicit list of everything that goes out in a release. Hoe uses this during packaging so that nothing embarrassing is picked up.

I'll expand more on this later since it seems to be a point of contention.

VERSION

Releases have versions and I've found it best for the version to be part of the code. You can use this during runtime in a multitude of ways. Hoe finds your version and uses it automatically during packaging.

Releasing in 1 easy step

```
% rake release VERSION=x.y.z
```

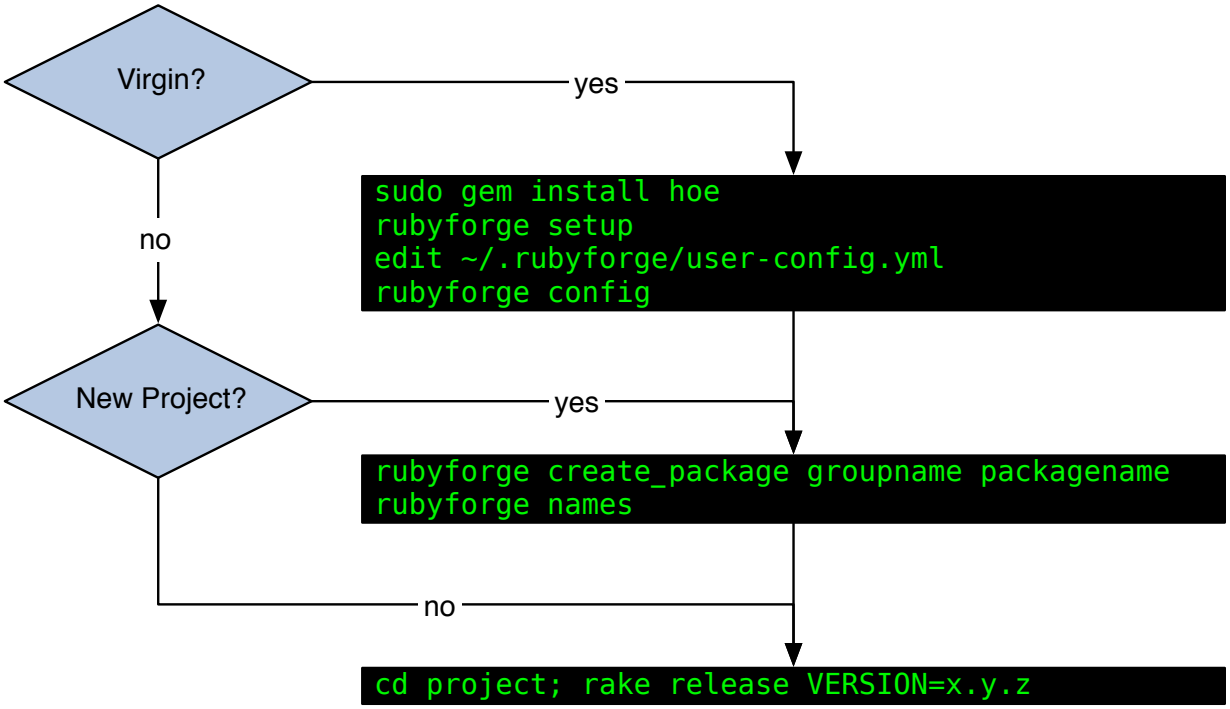
That really is all there is to it. With the `hoe-seattlerb` plugin, that branches the release in our perforce server, performs sanity checks to ensure the release has integrity, packages into gem and tarballs, uploads the packages to rubyforge, posts news of the release to rubyforge and my blog, and crafts an announcement email (future plugins will send the email directly).

That `VERSION=x.y.z` is there as a last-chance sanity check that you know what you're releasing. You'd be surprised how blurry eyed/brained you get at 3AM. This check helps a lot more than it should.

Setting Up Rubyforge

Assuming you're planning on releasing on rubyforge, you need to have rubyforge set up properly. Getting everything up and running the first time can be a bit of a PITA, but once you're done, you'll never have to do (most of) it again.

Everyone ***loves*** flowcharts! So enjoy:



Writing a Hoe Spec

Converting Hoe 1.x Specs

Not too much to do here. Basic steps to convert are:

+ Hoe.new becomes Hoe.spec.
+ Remove the internal require and version constant from Hoe.spec.
+ Remove the block argument (a bug/feature in 1.9 prevents it atm).
+ Make sure that you use `self.` for assignments to prevent assigning to a local variable.

Hoe 1.x Spec

```
require 'hoe'
require './lib/blah.rb'

Hoe.new('blah', Blah::VERSION) do |blah|
  blah.rubyforge_name = 'seattlerb'

  blah.developer 'Ryan Davis', 'ryand-ruby@zenspider.com'

  blah.extra_deps << 'whatevs'
end
```

Hoe 2.x Spec

```
require 'hoe'

Hoe.spec 'blah' do
  self.rubyforge_name = 'seattlerb'

  developer 'Ryan Davis', 'ryand-ruby@zenspider.com'

  extra_deps << 'whatevs'
end
```

Prettier, no?

From Scratch

The easiest way to get started with hoe is to use its included command-line tool `sow`:

```
% sow my_new_project
```

That will create a new directory `my_new_project` with a skeletal project inside. You need to edit the Rakefile with developer information in order to meet the minimum requirements of a working hoe-spec. You should also go fix all the things it points out as being labeled with "FIX" in the README.txt file.

Using Sow Templates

If you're planning on releasing a lot of packages and you've got certain recipes you like to have in your project, do note that sow uses a template directory and ERB to create your project. The first time you run sow it creates ~/.hoe_template. Make modifications there and every subsequent project will have those changes.

Extending Hoe with Plugins

Hoe has a flexible plugin system with the release of 2.0. This allowed Hoe to be refactored. That in and of itself was worth the effort. Probably more important is that it allows you to customize your projects' tasks in a modular and reusable way.

Using Hoe Plugins

Using a Hoe plugin is incredibly easy. Activate it by calling Hoe.plugin like so:

```
Hoe.plugin :minitest
```

This will activate the Hoe::Minitest plugin, attach it and load its tasks and methods into your hoe-spec. Easy-peasy!

Writing Hoe Plugins

A plugin can be as simple as:

```
module Hoe::Thingy
  attr_accessor :thingy

  def initialize_thingy # optional
    self.thingy = 42
  end

  def define_thingy_tasks
    task :thingy do
      puts thingy
    end
  end
end
```

Not terribly useful, but you get the idea. This example exercises both plugin methods (initialize_#{plugin} and define_#{plugin}_tasks and adds an accessor method to the Hoe instance.

How Plugins Work

Hoe plugins are made to be as simple as possible, but no simpler. They are modules defined in the `Hoe` namespace and have only one required method (`define_#{plugin}_tasks`) and one optional method (`initialize_#{plugin}`). Plugins can also define their own methods and they'll be available as instance methods to your hoe-spec. Plugins have 4 simple phases:

Loading

When Hoe is loaded the last thing it does is to ask rubygems for all of its plugins. Plugins are found by finding all files matching "hoe/*.rb" via installed gems or `$LOAD_PATH`. All found files are then loaded.

Activation

All of the plugins that ship with hoe are activated by default. This is because they're providing the same functionality that the previous Hoe was and without them, it'd be rather useless. Other plugins should be "opt-in" and are activated by:

```
Hoe::plugin :thingy
```

Put this _above_ your hoe-spec. All it does is add `:thingy` to `Hoe.plugins`. You could also deactivate a plugin by removing it from `Hoe.plugins` although that shouldn't be necessary for the most part.

Please note that it is **not** a good idea to have a plugin you're writing activate itself. Let developers opt-in, not opt-out. Just because someone needs the `:thingy` plugin on one project doesn't mean they need them on _all_ their projects.

Initialization

When your hoe-spec is instantiated, it extends itself all known plugin modules. This adds the method bodies to the hoe-spec and allows for the plugin to work as part of the spec itself. Once that is over, activated plugins have their **optional** define `initialize_#{plugin}` methods called. This lets them set needed instance variables to default values. Finally, the hoe-spec block is evaluated so that project specific values can override the defaults.

Task Definition

Finally, once the user's hoe-spec has been evaluated, all activated plugins have their `define_#{plugin}_tasks` method called. This method must be defined and it is here that you'll define all your tasks.

Questions & Counterpoints

"Why should I maintain a Manifest.txt when I can just write a glob?"

> manifest <super>2</super> |ˈmønəˈfɛst| |
ˈmanɪfɛst|
> noun
> a document giving comprehensive details of a ship and its cargo and other contents, passengers, and crew for the use of customs officers.

Imagine, you're a customs inspector at the Los Angeles Port, the world's largest import/export port. A large ship filled to the brim pulls up to the pier ready for inspection. You walk up to the captain and his crew and ask "what is the contents of this fine ship today" and the captain answers "oh... whatever is inside". The mind boggles. There is no way in the world that a professionally run ship would ever run this way and there is no way that you should either.

Professional software releases know _exactly_ what is in them, amateur releases do not. "Write better globs" is the response I often hear. I consider myself and the people I work with to be rather smart people and if we get them wrong, chances are you will too. How many times have you peered under the covers and seen .DS_Store, emacs backup~ files, vim vm files and other files completely unrelated to the package? I have far more times than I'd like.

"Why not just write gemspecs?"

Short answer: I've done that and it is way too much work.

Medium answer: It isn't DRY. All my projects have a history file, a readme, some code with a version string, etc. Why should I duplicate all of that information into the gem spec when I can have code do it for me automatically? It is less error prone as a result. I screw up things, hoe doesn't.

Long answer: See that hoe spec above for the fictional "blah" project? This is the corresponding gem spec in all its glory (as cleaned up as I can/am willing to get it):

```
# -*- encoding: utf-8 -*-

Gem::Specification.new do |s|
  s.name              = "blah"
  s.version           = "1.0.0"

  s.authors           = ["Ryan Davis"]
  s.description       = "..."
  s.email             = ["ryand-ruby@zenspider.com"]
  s.executables       = ["blah"]
  s.extra_rdoc_files  = [...]
  s.files             = [...]
  s.homepage          = "..."
  s.rdoc_options      = ["--main", "README.txt"]
  s.rubyforge_project = "seattlerb"
  s.summary           = "..."
  s.test_files        = [...]

  s.cert_chain        = ["/Users/ryan/.gem/gem-
public_cert.pem"]
  s.signing_key       = "/Users/ryan/.gem/gem-private_key.pem"

  s.add_runtime_dependency(%q<whatevs>, [">= 0"])
end
```

"What about (newgem|bones|echoe|joe|gemify|...)?"

Smoke 'em if ya got 'em.

All I can really say is that Hoe works _really_ well for me and a lot of others. As of this writing, a simple grep across all current-version gems show that Hoe is used by 1375 (or 27.5%) of the 4993 published gems. Some of these were probably created by other packages that wrap up Hoe (like newgem), but further analysis was not attempted to differentiate actual origin. if they use Hoe, then they were counted as Hoe.