# The Dark Art of Rails Plugins

reevoo.com

James Adam

# Anatomy of a plugin

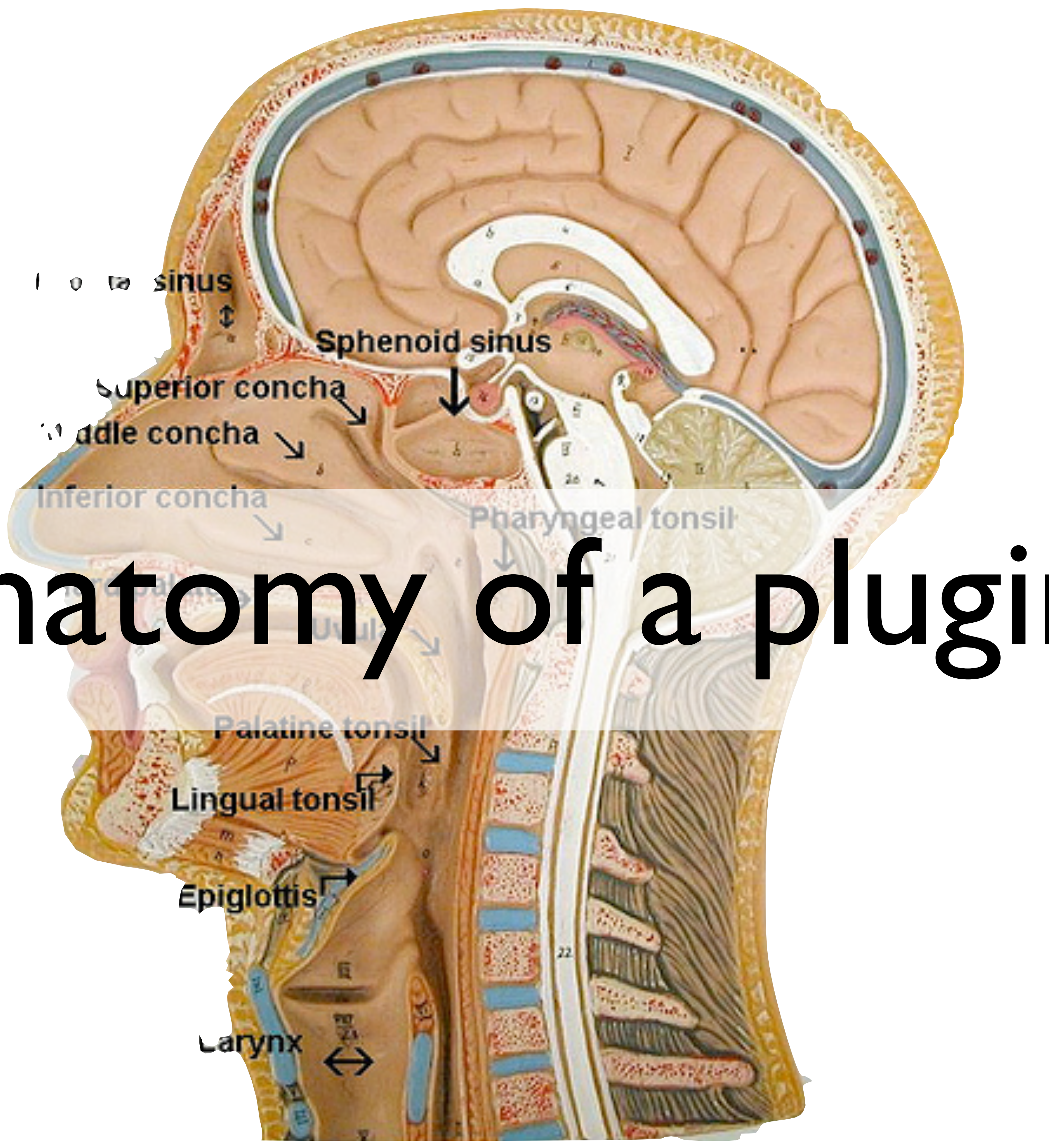
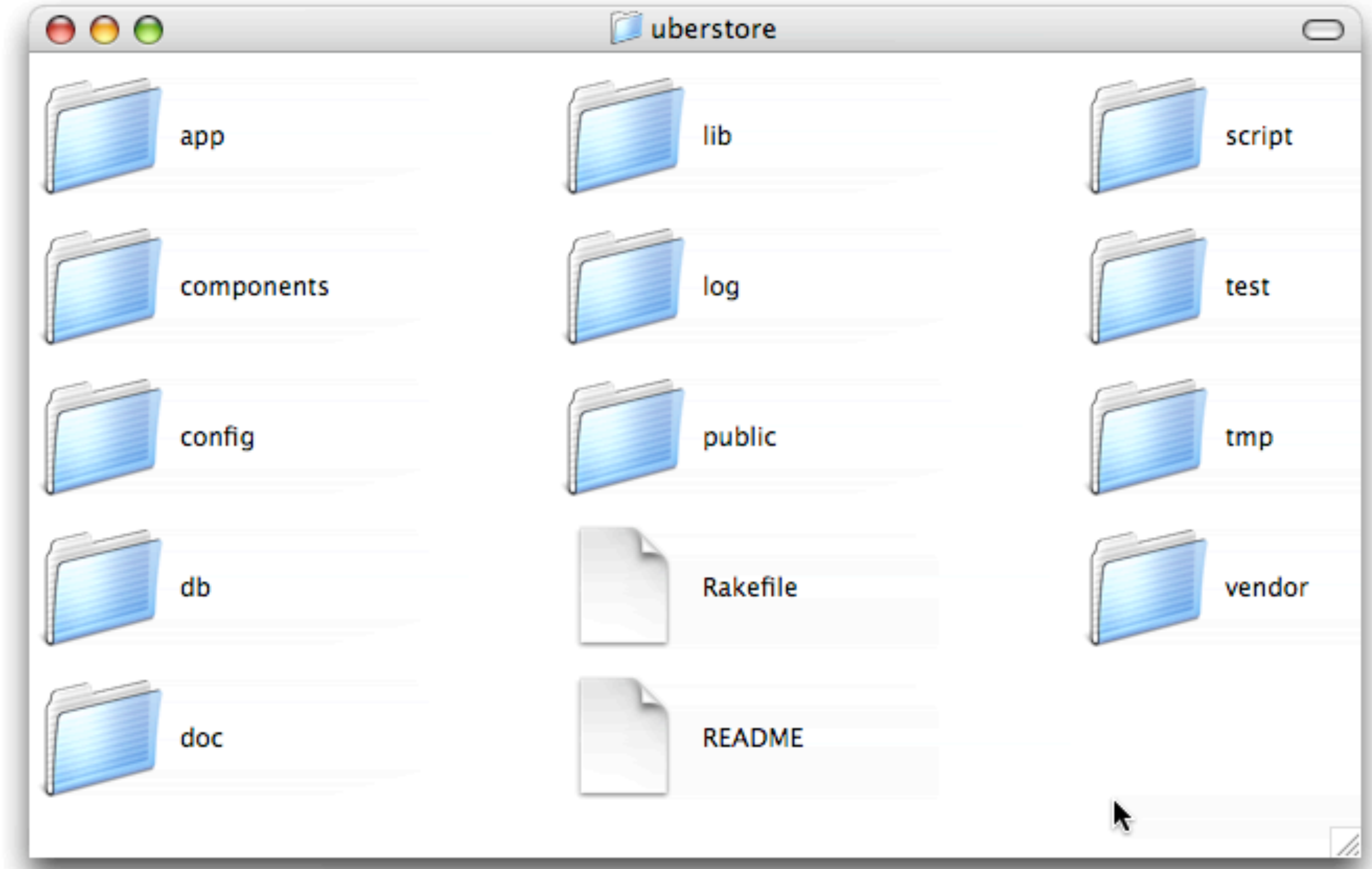


Photo: http://flickr.com/photos/guccibear2005/206352128/


uberstore
app
lib
script
components
log
test
config
public
tmp
db
Rakefile
vendor
doc
README

lib

## lib

- added to the `$LOAD_PATH`
- classes auto-loaded via `Dependencies` magic
- order determined by `config.plugins`

# init.rb

# init.rb

- evaluated near the end of rails initialization
- evaluated in order of `config.plugins`
- special variables available
  - `config`, `directory`, `name` - see source of `Rails::Plugin`

[un]install.rb

tasks

generators

test

## Writing Plugins

# Sharing Code

lib

tasks

## Adding new behaviour

## What we're aiming for

```
class Person
end

class Programmer < Person
end

class ProjectManager < Person
end

p = Programmer.new
p.hello # => "hi!"

ProjectManager.is_friendly? # => true
```

## Modules

```
module Friendly
  def hello
    "hi from #{self}"
  end
end
```

```ruby
require 'friendly'

class Person
  include Friendly
end

alice = Programmer.new
alice.hello
# => "hi from #<Programmer#123>"
```

## Methods in classes...

```ruby
class Person
  def hello
    "hi from #{self}"
  end
end
```

## ... and in modules

```
module Friendly
  def hello
    "hi from #{self}"
  end
end
```

## Defining class methods...

```
class Person
  def self.is_friendly?
    true
  end
end
```

## ... and in modules?

```
module Friendly
  def self.is_friendly?
    true
  end

  def hello
    "hi from #{self}"
  end
end
```

## Not quite :(

```
class Person
  include Friendly
end

Person.is_friendly?
# ~> undefined method `is_friendly?'
for Person:Class (NoMethodError)
```

## It’s all about `self`

```ruby
module Friendly
  def self.is_friendly?
    true
  end
end
```

## Try this instead

```
module Friendly::ClassMethods
  def is_friendly?
    true
  end
end

class Person
  extend Friendly::ClassMethods
end

Person.is_friendly? # => true
```

## Mixing in Modules

```
class Person
  include AnyModule
  # adds to class definition
end

class Person
  extend AnyModule
  # adds to the object (self)
end
```

## Some other ways:

```
Person.instance_eval do
  def greetings
    "hello via \
     instance_eval"
  end
end
```

## Some other ways:

```ruby
class << Person
  def salutations
    "hello via \
     class << Person"
  end
end
```

```ruby
module ActsAsFriendly
  module ClassMethods
    def is_friendly?
      true
    end
  end

  def hello
    "hi from #{self}!"
  end
end

Person.send(:include, ActsAsFriendly)
Person.extend(ActsAsFriendly::ClassMethods)
```

## included

```ruby
module B
  def self.included(base)
    puts "B included into #{base}!"
  end
end

class A
  include B
end
# => "B included into A!"
```

## extended

```ruby
module B
  def self.extended(base)
    puts "#{base} extended by B!"
  end
end

class A
  extend B
end
# => "A extended by B!"
```

```ruby
module ActsAsFriendly
  def self.included(base)
    base.extend(ClassMethods)
  end

  module ClassMethods
    def is_friendly?
      true
    end
  end

  def hello
    "hi from #{self}!"
  end
end

Person.send(:include, ActsAsFriendly)
```

```ruby
module ActsAsFriendly
  def self.included(base)
    base.extend(ClassMethods)
  end

  module ClassMethods
    def is_friendly?
      true
    end
  end

  def hello
    "hi from #{self}!"
  end
end

Person.send(:include, ActsAsFriendly)
```

```ruby
class EvilBoss < Person
end

EvilBoss.is_friendly? # => true
```

## Showing restraint...

- maybe we only want to apply it to particular classes
- particularly if we’re going to change how the class behaves (see later...)

## ... using class methods

- Ruby class definitions **are** code
- So, `has_many` is a *class method*

# Class methods!

```
class BlogPost < ActiveRecord::Base

  has_many :comments
  validates_presence_of :title

end
```

## Class methods!

```
class BlogController <
                ApplicationController

  before_filter :load_blog_posts

end
```

## Class methods!

```ruby
class Person
  attr_reader :name
end
```

## Self in class definitions

```
class SomeClass
  puts self
end
# >> SomeClass
```

## Calling methods

```ruby
class SomeClass
  def self.greetings
    "hello"
  end
  greetings
end
# >> hello
```

```ruby
module AbilityToFly
  def fly!
    true
  end
  # etc...
end

class Person
  def self.has_powers
    include AbilityToFly
  end
end
```

```ruby
class Villain < Person
end

class Hero < Person
  has_powers
end


clark_kent = Hero.new
clark_kent.fly! # => true

lex_luthor = Villain.new
lex_luthor.fly! # => NoMethodError
```

```ruby
Villain.has_powers
lex.fly! # => true
```

```
module MyPlugin
  def acts_as_friendly
    include MyPlugin::ActsAsFriendly
  end

  module ActsAsFriendly
    def self.included(base)
      base.extend(ClassMethods)
    end

    module ClassMethods
      def is_friendly?
        true
      end
    end

    def hello
      "hi from #{self}"
    end

  end # of ActsAsFriendly
end # of MyPlugin

Person.extend(MyPlugin)
```

```ruby
module MyPlugin
  def acts_as_friendly
    include MyPlugin::ActsAsFriendly
  end

  module ActsAsFriendly
    def self.included(base)
      base.extend(ClassMethods)
    end

    module ClassMethods
      def is_friendly?
        true
      end
    end

    def hello
      "hi from #{self}"
    end

  end # of ActsAsFriendly
end # of MyPlugin

Person.extend(MyPlugin)
```

```
module MyPlugin
  def acts_as_friendly
    include MyPlugin::ActsAsFriendly
  end

  module ActsAsFriendly
    def self.included(base)
      base.extend(ClassMethods)
    end

    module ClassMethods
      def is_friendly?
        true
      end
    end

    def hello
      "hi from #{self}"
    end

  end # of ActsAsFriendly
end # of MyPlugin

Person.extend(MyPlugin)
```

```ruby
class Grouch < Person
end

oscar = Grouch.new
oscar.hello # => NoMethodError

class Hacker < Person
  acts_as_friendly
end

Hacker.is_friendly? # => true
james = Hacker.new
james.hello # => “hi from #<Hacker:0x123>”
```

```
Person.extend(MyPlugin)
```

```ruby
ActiveRecord::Base.extend(MyPlugin)
```

## Extending Rails

RAILS

## acts_as_archivable

- when a record is deleted, save a YAML version. Just in case.
- It’s an odd example, but bear with me.

## Archivable Module

```
module Archivable
  def archive_to_yaml
    File.open("#{id}.yml", 'w') do |f|
      f.write self.to_yaml
    end
  end
end

ActiveRecord::Base.send(:include,
                        Archivable)
```

# Redefine via a module

```
module Archivable
  def archive_to_yaml
    File.open("#{id}.yml") # ...etc...
  end

  def destroy # redefine destroy!
    connection.delete %{
      DELETE FROM #{table_name}
      WHERE id = #{self.id}
    }
    archive_to_yaml
  end
end
```

# Redefine via a module

```
ActiveRecord::Base.send(:include,
                        Archivable)

class Thing < ActiveRecord::Base
end

t = Thing.find(:first)

t.destroy # => no archive created :'(
```

# Redefining in the class

```
class ActiveRecord::Base
  def destroy
    # Actually delete the record
    connection.delete %{
      DELETE FROM #{table_name}
      WHERE id = #{self.id}
    }

    # call our new method
    archive_to_yaml
  end
end
```

# ...it's ~~evil~~ naughty

- ties our new functionality to ActiveRecord, in this example
  - maybe we want to add this to DataMapper? Or Sequel? Or Ambition?
- **What if Rails changes?**

## What we want

- Our new behaviour should be triggered when `destroy` is called
- The record should still be removed from the database
- We shouldn’t need to reimplement the existing behaviour

## alias_method

```
def hello
  "hi!"
end

alias_method :greetings,
             :hello

greetings # => "hi!"
```

## alias_method

```
alias_method :original_destroy,
             :destroy

def new_destroy
  original_destroy

  archive_to_yaml
end

alias_method :destroy,
             :new_destroy
```

```
module Archivable
  alias_method :original_destroy, :destroy

  def new_destroy
    original_destroy
    archive_to_yaml
  end

  alias_method :destroy, :new_destroy
end

# ~> undefined method `destroy' for
     module `Archivable'
```

```ruby
module Archivable
  def self.included(base)
    base.class_eval do
      alias_method :original_destroy, :destroy
      alias_method :destroy, :new_destroy
    end
  end

  def archive_to_yaml
    File.open("#{id}.yml") # ...
  end

  def new_destroy
    original_destroy
    archive_to_yaml
  end
end

ActiveRecord::Base.send(:include, Archivable)
```

```ruby
class Thing < ActiveRecord::Base
end

t = Thing.find(:first)

t.destroy # => archive created!
```

## alias_method again

```
alias_method :destroy_without_archiving,
             :destroy

def destroy_with_archiving
  destroy_without_archiving
  archive_to_yaml
end

alias_method :destroy,
             :destroy_with_archiving
```

# alias_method_chain

```ruby
def destroy_with_archiving
  destroy_without_archiving
  archive_to_yaml
end

alias_method_chain :destroy,
                   :archiving
```

```ruby
module Archivable
  def self.included(base)
    base.class_eval do
      alias_method_chain :destroy, :archiving
    end
  end

  def archive_to_yaml
    File.open("#{id}.yml", "w") do |f|
      f.write self.to_yaml
    end
  end

  def destroy_with_archiving
    destroy_without_archiving
    archive_to_yaml
  end
end

ActiveRecord::Base.send(:include, Archivable)
```

## So adding up everything

- use `extend` to add class method
- include the new behaviour by including a module when class method is called
- use `alias_method_chain` to wrap existing method

```ruby
module ActsAsArchivable
  def acts_as_archivable
    include ActsAsArchivable::Behaviour
  end

  module Behaviour
    def self.included(base)
      base.class_eval do
        alias_method_chain :destroy, :archiving
      end
    end

    def archive_to_yaml
      File.open("#{id}.yml") # ...
    end

    def destroy_with_archiving
      destroy_without_archiving
      archive_to_yaml
    end
  end
end

ActiveRecord::Base.extend(ActsAsArchivable)
```

```ruby
module ActsAsArchivable
  def acts_as_archivable
    include ActsAsArchivable::Behaviour
    alias_method_chain :destroy, :archiving
  end

  module Behaviour
    def archive_to_yaml
      File.open("#{id}.yml") # ...
    end

    def destroy_with_archiving
      destroy_without_archiving
      archive_to_yaml
    end
  end
end

ActiveRecord::Base.extend(ActsAsArchivable)
```

```
module ActsAsArchivable
  def acts_as_archivable
    include ActsAsArchivable::Behaviour
    alias_method_chain :destroy, :archiving
  end

  module Behaviour
    def archive_to_yaml
      File.open("#{id}.yml") # ...
    end

    def destroy_with_archiving
      destroy_without_archiving
      archive_to_yaml
    end
  end
end

ActiveRecord::Base.extend(ActsAsArchivable)
```

```ruby
class Thing < ActiveRecord::Base
end

t1 = Thing.create!
t1.destroy # => normal destroy called
Thing.count # => 0
Dir["*.yml"] # => []

class PreciousThing < ActiveRecord::Base
  acts_as_archivable
end
t2 = PreciousThing.create!
t2.destroy
PreciousThing.count # => 0
Dir["*.yml"] # => ["1.yml"]
```

## Plugin Tips

Photo: http://flickr.com/photos/jreed/322057793/

Tip #1

## Package your code

- ...in a module
- domain name, nickname, quirk

```
module Lazyatom
  module ActsAsHasselhoff
    # ...
  end
end
```

Tip #2

## Developing plugins

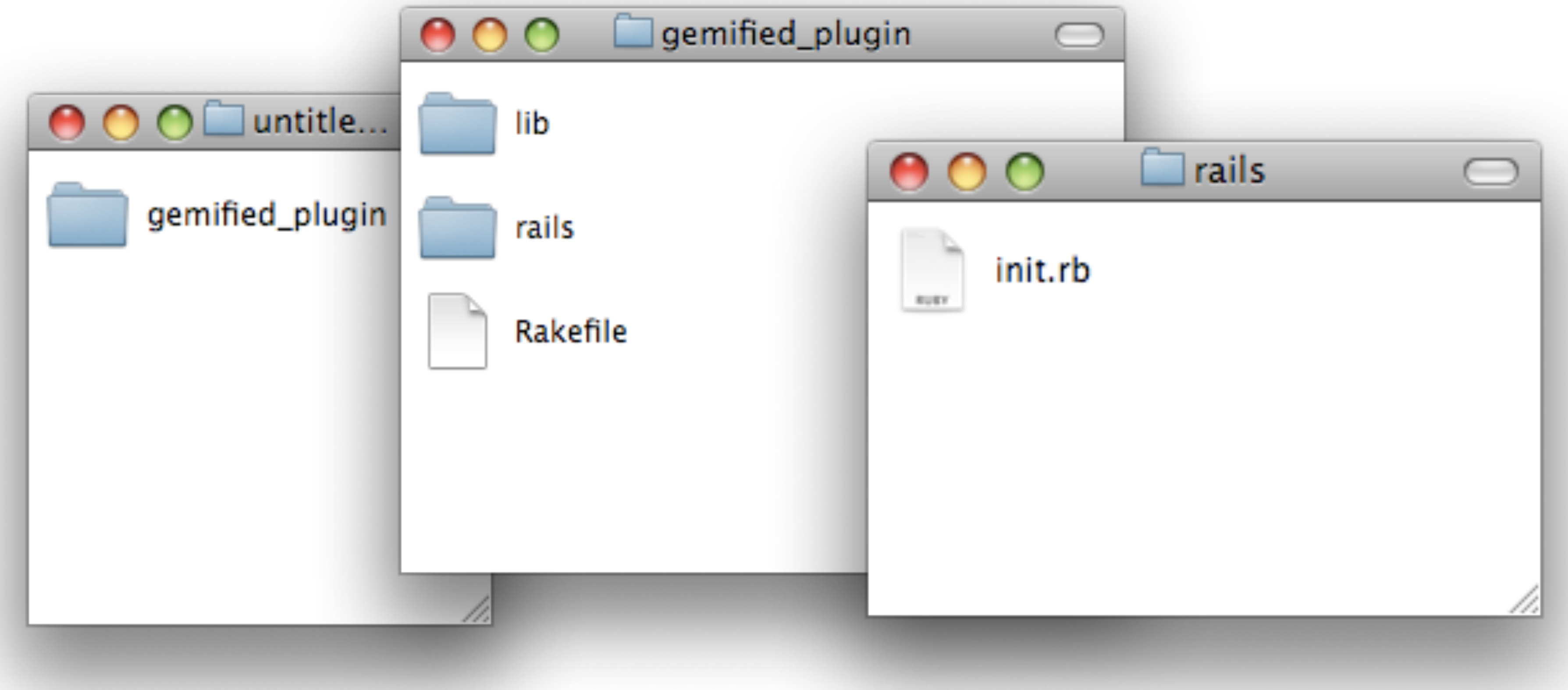
- Dependencies.load_once_paths
- config/environment/development.rb

```
config.after_initialize do
  Dependencies.load_once_paths.
    delete_if do |path|
      path =~ /vendor\/plugins/
    end
end
```

Tip #3

## Gems as plugins

- coming in Rails 2.1 (it's in r9101)
- add `rails/init.rb` to your gem



- `require "rubygems"` in environment.rb

# Thanks!

lazyatom.com/plugins