Data-Driven Classes in Ruby

Michael Granger and David McCorkhill

O’Reilly Open Source Convention
July 7 - 11, 2003
Conventions

- Examples use some features only found in Ruby 1.8
- Error-checking in example code has been omitted for brevity
Overview
What are “data-driven” classes?

- All classes are data-driven in some sense.
- Data-driven classes are classes defined by dynamic information.
  - Classes which modify or create their behavior based on state (data).
  - Classes not (completely) defined by source or object code
Motivations: Why might I want to use them?
Motivations

- Data-driven techniques minimize the impact of new or changing data structures and APIs
- Expressing functionality with data allows for run-time changes
- Allows code to be managed in parceled chunks
Minimize impact

• Databases
  – Field names
  – New tables
  – Changing datatypes
• Web Services
  – Remote service encapsulation
  – Reflection of changes in remote APIs
Allows run-time changes

- Mutating Parsers
  - Parsers which change their grammar as they parse
- Unit Testing
  - Parse a specification to generate testing code
- Command Interpreters
  - Shells, command line interfaces
  - Aliases, user functions, macros, etc.
Manage code in chunks

• Games
  – Game worlds can be highly dynamic and long-running, requiring frequent changes while running.
  – Creators are not necessarily coders.
• Conformance testing
  – Customer creates tests via a coding abstraction.
Why Ruby?
Why Ruby?

- A dynamic language
  - Much easier to modify execution on the fly
  - Loose typing
- Its idiom already encourages thinking about functionality as data:
  - `attr_accessor :foo`
  - `Class::new`
  - `Object#method`
Why Ruby? (cont.)

• Good support for code-writing code and runtime extension.
  – Singleton methods allow modification on an instance-by-instance basis
  – class << obj
  – Mixins allow even the data-driven behavior itself to be abstracted and reused in other code.
Techniques
Three Techniques for Driving Programs with Data

- Choosing a class based on data
- Reflecting data structure in methods
- Using metaclasses, code-writing code, and dynamic factories
Choosing a Class Based on Data

- Good for situations in which a back end is specified by a small amount of data such as a configuration item or a command-line option
  - DSNs
  - URIs (http, ftp, nntp, etc.)
  - Encryption Algorithms (blowfish, RSA, MD5, etc.)
  - Document Types (RTF, XML, YAML, etc.)
- Use a Factory Method to provide an interface for picking a class based on hinge data
Factory Methods

• Add a method to a base class that can instantiate subclasses based on hinge data
• Register subclasses with the base class through the inheritance callback
• Associate each subclass with one or more variants of its name or other data for flexibility
• Make variants as close as possible to the expected input for ease of use.
Factory Method Example 1

class Document
  @derivatives = {}
  class << self
    attr_reader :derivatives
  end

  def self::inherited( klass )
    Document::derivatives[ klass.name ] = klass
  end

  def self::create( name, *args )
    Document::derivatives[ name ].new( *args )
  end
end
Factory Method Example 1

class Document
  @derivatives = {}

class << self
  attr_reader :derivatives
end

def self::inherited( klass )
  Document::derivatives[ klass.name ] = klass
end

def self::create( name, *args )
  Document::derivatives[ name ].new( *args )
end
end
Factory Method Example 1

class Document
  @derivatives = {}

  class << self
    attr_reader :derivatives
  end

  def self::inherited( klass )
    Document::derivatives[ klass.name ] = klass
  end

  def self::create( name, *args )
    Document::derivatives[ name ].new( *args )
  end
end
class Document
  @@derivatives = {}
  class << self
    attr_reader :derivatives
  end

  def self::inherited( klass )
    Document::derivatives[klass.name] = klass
  end

  def self::create( name, *args )
    Document::derivatives[name].new(*args)
  end
end
Factory Method Example 1

class Document
  @derivatives = {}
  class << self
    attr_reader :derivatives
  end

  def self::inherited( klass )
    Document::derivatives[ klass.name ] = klass
  end

  def self::create( name, *args )
    Document::derivatives[ name ].new( *args )
  end
end
Factory Method Example 1

class Document
  [...]
end

class XmlDocument < Document; end
class HtmlDocument < XmlDocument; end
class PdfDocument < Document; end

p Document::create( "XmlDocument" )
=> #<XmlDocument:0x40290628>

p Document::create( "HtmlDocument" )
=> #<HtmlDocument:0x402905d8>
Factory Method Example 2

class Document
    [...]  
    def self::inherited( klass )
        Document::derivatives[ klass.name.downcase ] = klass
        ext = klass.name.gsub(/document/i, '')
        Document::derivatives[ ext.downcase ] = klass
    end

    def self::create( name, *args )
        Document::derivatives[ name.downcase ].new( *args )
    end
end
class Document
  
  def self::inherited( klass )
      Document::derivatives[ klass.name.downcase ] = klass
      ext = klass.name.gsub(/document/i, '')
      Document::derivatives[ ext.downcase ] = klass
  end

  def self::create( name, *args )
      Document::derivatives[ name.downcase ].new( *args )
  end

end
Factory Method Example 2

class Document
    [...] end

class XmlDocument < Document; end
class HtmlDocument < XmlDocument; end
class PdfDocument < Document; end

p Document::create( 'pdf' )
=> #<PdfDocument:0x4028fde0>
class Document
    
    def self.create(name, *args)
        derivs = Document::derivatives
        ext = File::extname(name)[1..-1]

        if derivs.key?(name.downcase)
            return derivs[name.downcase].new(*args)
        elsif derivs.key?(ext)
            return derivs[ext].new(*args)
        end
    end
end
Factory Method Example 2

class Document
    [...]
end

class XmlDocument < Document; end
class HtmlDocument < XmlDocument; end
class PdfDocument < Document; end

p Document::create( ‘presentation.pdf’ )
=> #<PdfDocument:0x4028f96c>
Reflecting data structure in methods

- Good for situations when you want a more granular approach than selecting a whole class
  - When data members are fairly similar
  - Load only the methods you need, when you need them
  - Allow one generic class to handle many different data structures
Simple Method Autoloading

- Set up a generic class which is responsible for encapsulating a data structure
- Do not add specific accessors or mutators so the class can be used for more than one data structure
- Catch calls to missing methods
- If the name of the method being called matches a part of the encapsulated data, act as an accessor or mutator
class Adapter
  def initialize( row )
    @row = row
  end

  def method_missing( sym, *args )
    name = sym.to_s.sub(/=$/, '')
    super unless @row.key?( name )
    if /=$/ =~ sym.to_s
      return @row[ name ] = args[0]
    else
      return @row[ name ]
    end
  end
end
Simple Autoloading Example

```ruby
class Adapter
  def initialize( row )
    @row = row
  end

  def method_missing( sym, *args )
    name = sym.to_s.sub(/=$/, '')
    super unless @row.key?( name )
    if /=$/ =~ sym.to_s
      return @row[ name ] = args[0]
    else
      return @row[ name ]
    end
  end
end
```
Simple Autoloading Example

class Adapter
  def initialize( row )
    @row = row
  end

  def method_missing( sym, *args )
    name = sym.to_s.sub(/=$/, '')
    super unless @row.key?( name )
    if /=$/ =~ sym.to_s
      return @row[ name ] = args[0]
    else
      return @row[ name ]
    end
  end
end
Simple Autoloading Example

class Adapter
  def initialize( row )
    @row = row
  end

  def method_missing( sym, *args )
    name = sym.to_s.sub(/=$/, '')
    super unless @row.key?( name )
    if /=$/ =~ sym.to_s
      return @row[ name ] = args[0]
    else
      return @row[ name ]
    end
  end
end
Simple Autoloading Example

class Adapter
  [...]
end

ad = Adapter::new( 'firstname' => 'Roger',
  'lastname' => 'Pollack',
  'nick' => '(unknown)' )

p "%s, %s: %s" % [ ad.lastname, ad.firstname, ad.nick ]
=> "Pollack, Roger: (unknown)"

ad.nick = "Mr. Slippery"

p "%s, %s: %s" % [ ad.lastname, ad.firstname, ad.nick ]
=> "Pollack, Roger: Mr. Slippery"
Metaclases, Code-Writing Code, and Dynamic Factories

• The highest levels of abstraction
• Useful when the specific behavior and structure of a part of the system must be deferred until as late as possible
  – Remote APIs (e.g., web services)
  – Database schemas
  – Highly volatile code (e.g., ORPGs, genetic programming, etc.)
Code-writing Code - Variant on Autoloading

- Can be used to gain the benefits of autoloading but with minimal cost to execution speed.
- Instead of dispatching to a single method with a key, generate the missing method on the first call.
- Can be used per-class when the data structure is the same across instances, or per-instance (via singleton methods) when it changes instance-by-instance.
class Adapter
    [...]  
    def method_missing( sym, *args )
        name = sym.to_s.sub(/=$/, '')
        super unless @row.key?( name )

        if /=$/ =~ sym.to_s
            meth =
                %{def #{sym}(arg); @row['#{name}'] = arg; end}
        else
            meth = %{def #{sym}; @row['#{name}']; end}
        end

        self.class.module_eval meth, __FILE__, __LINE__
        self.send( sym, *args )
    end
end
class Adapter

[...]

def method_missing( sym, *args )
  name = sym.to_s.sub(/=$/, '')
  super unless @row.key?( name )

  if /=$/ =~ sym.to_s
    
    meth =
    %{def #{sym}(arg); @row['#{name}'] = arg; end}
  else
    
    meth = %{def #{sym}; @row['#{name}']; end}
  end

  self.class.module_eval meth, __FILE__, __LINE__

  self.send( sym, *args )

end
end
def method_missing( sym, *args )
    name = sym.to_s.sub(/=$/, '')
    super unless @row.key?( name )

    if /=/$/ =~ sym.to_s
        meth =
            %{def #{sym}(arg); @row['#{name}'] = arg; end}
    else
        meth = %{def #{sym}; @row['#{name}']; end}
    end

    self.class.module_eval meth, __FILE__, __LINE__
    self.send( sym, *args )
end
end
Code-Writing Code - Autoloading

class Adapter
  [...]
end

ad = Adapter::new( 'firstname' => 'Roger',
  'lastname' => 'Pollack',
  'nick' => '(unknown)' )

p ad.lastname
"Pollack"

p ad.class.instance_methods( false )
["method_missing", "lastname"]
Code-writing Code - Test Generation

- Condense tests into testing datasets
  - Each test is a tuple with input $\Rightarrow$ expected output
  - Test Arrays are stored in a Hash of Arrays, keyed by method name as a symbol
- Create a test method for each item in the dataset
- Datasets aren’t limited to code
  - Can be read from a database
  - Parsed from a specially-formatted specification, etc.
## Code-Writing Code - Test Generation

```ruby
require 'test/unit'
require 'Soundex'

class SoundexTestCase < Test::Unit::TestCase
  include Text::Soundex
  TestMatrix = [
    [ "Euler", "E460" ],
    [ "Ellery", "E460" ],
    [ "Gauss", "G200" ],
    [ "Ghosh", "G200" ],
    [ "Hilbert", "H416" ],
    [ "Heilbronn", "H416" ],
    [ "Knuth", "K530" ],
    [ "Kant", "K530" ],
    [ "Lloyd", "L300" ],
    [ "Ladd", "L300" ],
  ]

  TestMatrix.each_with_index {|tuple,i|methname = "test_%03d_%s" % [i, tuple[0]]
eval {%
def #{methname}
rval = nil
assert_nothing_raised {
rval = soundex("#{tuple[0]}")
}
assert_equal "#{tuple[1]}", rval,
"Soundex of '#{tuple[0]}'
end
  }
  end
```

Data-Driven Classes in Ruby

Code-Writing Code - Test Generation

```ruby
require 'test/unit'
require 'Soundex'

class SoundexTestCase < Test::Unit::TestCase
  include Text::Soundex
  TestMatrix = [
    [ "Euler", "E460" ],
    [ "Ellery", "E460" ],
    [ "Gauss", "G200" ],
    [ "Ghosh", "G200" ],
    [ "Hilbert", "H416" ],
    [ "Heilbronn", "H416" ],
    [ "Knuth", "K530" ],
    [ "Kant", "K530" ],
    [ "Lloyd", "L300" ],
    [ "Ladd", "L300" ],
  ]

  TestMatrix.each_with_index {|tuple,i|,
    methname = "test_%03d_%s" % [ i, tuple[0] ]
    eval {,
      def #{methname}
        rval = nil
        assert_nothing_raised {
          rval = soundex("#{tuple[0]}")
        }
        assert_equal "#{tuple[1]}", rval,
                      "Soundex of '#{tuple[0]}'"
      end
    }
  }
end
```
require 'test/unit'
require 'Soundex'

class SoundexTestCase < Test::Unit::TestCase
  include Text::Soundex

  TestMatrix = [
    [ "Euler",   "E460" ],
    [ "Ellery",  "E460" ],
    [ "Gauss",   "G200" ],
    [ "Ghosh",   "G200" ],
    [ "Hilbert", "H416" ],
    [ "Heilbronn","H416" ],
    [ "Knuth",   "K530" ],
    [ "Kant",    "K530" ],
    [ "Lloyd",   "L300" ],
    [ "Ladd",    "L300" ],
  ]

  TestMatrix.each_with_index { |tuple,i|
    methname = "test_%03d_%s" % [ i, tuple[0] ]
    eval %{def #{methname}
      rval = nil
      assert_nothing_raised {
        rval = soundex("#{tuple[0]}")
      }
      assert_equal "#{tuple[1]}", rval,
      "Soundex of '#{tuple[0]}'")
    end
  }
end
Data-Driven Classes in Ruby

```ruby
require 'test/unit'
require 'Soundex'

class SoundexTestCase < Test::Unit::TestCase
  include Text::Soundex

  TestMatrix = [
    [ "Euler", "E460" ],
    [ "Ellery", "E460" ],
    [ "Gauss", "G200" ],
    [ "Ghosh", "G200" ],
    [ "Hilbert", "H416" ],
    [ "Heilbronn", "H416" ],
    [ "Knuth", "K530" ],
    [ "Kant", "K530" ],
    [ "Lloyd", "L300" ],
    [ "Ladd", "L300" ],
  ]

  TestMatrix.each_with_index{|tuple,i|}
    methname = "test_%03d_%s" % [ i, tuple[0] ]
    eval %{def #{methname}
      rval = nil
      assert_nothing_raised {
        rval = soundex("#{tuple[0]}")
      }
      assert_equal "#{tuple[1]}", rval,
      "Soundex of '#{tuple[0]}'"
    end}
  end
end
```
require 'test/unit'
require 'Soundex'

class SoundexTestCase < Test::Unit::TestCase

include Text::Soundex

TestMatrix = [
  [ "Euler", "E460" ],
  [ "Ellery", "E460" ],
  [ "Gauss", "G200" ],
  [ "Ghosh", "G200" ],
  [ "Hilbert", "H416" ],
  [ "Heilbronn", "H416" ],
  [ "Knuth", "K530" ],
  [ "Kant", "K530" ],
  [ "Lloyd", "L300" ],
  [ "Ladd", "L300" ],
]

TestMatrix.each_with_index { |tuple, i|

  methname = "test_%03d_%s" % [ i, tuple[0] ]

  eval %{,
    def #{methname}
      rval = nil
      assert_nothing_raised {
        rval = soundex("#{tuple[0]}")
      }
      assert_equal "#{tuple[1]}", rval,
      "Soundex of '#{tuple[0]}'"
    end
  }, nil, __FILE__, __LINE__
}

end
require 'test/unit'
require 'Soundex'

class SoundexTestCase < Test::Unit::TestCase
  include Text::Soundex
  TestMatrix = [
  [ "Euler", "E460" ],
  [ "Ellery", "E460" ],
  [ "Gauss", "G200" ],
  [ "Ghosh", "G200" ],
  [ "Hilbert", "H416" ],
  [ "Heilbronn", "H416" ],
  [ "Knuth", "K530" ],
  [ "Kant", "K530" ],
  [ "Lloyd", "L300" ],
  [ "Ladd", "L300" ],
  ]

  TestMatrix.each_with_index { |tuple,i| 
    methname = "test_%03d_%s" % [i, tuple[0]]
    eval %{def #{methname}
      rval = nil
      assert_nothing_raised {
        rval = soundex("#{tuple[0]}")
      }
      assert_equal "#{tuple[1]}", rval,
        "Soundex of '#{tuple[0]}'"
    end
  }, nil, __FILE__, __LINE__
end
Code-Writing Code - Test Generation (output)

galendril pts/9 [~/source/ruby/OSCON2003]
57/535 {0}$ ruby code/autotest1.rb
Loaded suite code/autotest1
Started
............
Finished in 0.005167 seconds.

12 tests, 24 assertions, 0 failures, 0 errors
Dynamic Factories

• A factory object is given a meta-data description of a class
• …or fetches one from a given resource
• The factory generates and returns an anonymous wrapper class that encapsulates the structure of the data
• A code glossary can be used to provide a “parts bin” for assemblage of the class
Dynamic Factories - A Table Adapter Factory

```ruby
require 'dbi'

class TableAdapterFactory
  class Adapter
    Glossary = {
      :pkey_select => "select * from %s where %s = ?",
    }
    
    def self::lookup( pkey )
      rowdata = nil
      qry = Glossary[:pkey_select] % [table, primaryKey]
      self.dbh.prepare( qry ) { |sth|
        sth.execute( pkey )
        rowdata = sth.fetch.to_h
      }
      new( rowdata )
    end
  end
end
```

[...]

require 'dbi'

class TableAdapterFactory
  class Adapter
    Glossary = {
      :pkey_select => "select * from %s where %s = ?",
    }

    def self::lookup( pkey )
      rowdata = nil
      qry = Glossary[:pkey_select] % [table, primaryKey]
      selfdbh.prepare( qry ) { |sth|
        sth.execute( pkey ) { |sth|
          rowdata = sth.fetch.to_h
        }
        new( rowdata )
      }
      new( rowdata )
    end
  end
[
...]

Dynamic Factories - A Table Adapter Factory
Dynamic Factories - A Table Adapter Factory

```ruby
require 'dbi'

class TableAdapterFactory
  class Adapter
    Glossary = {
      :pkey_select => "select * from %s where %s = ?",
    }

    def self::lookup( pkey )
      rowdata = nil
      qry = Glossary[:pkey_select] % [table, primaryKey]
      self.dbh.prepare( qry ) { |sth|
        sth.execute( pkey )
        rowdata = sth.fetch.to_h
      }
      new( rowdata )
    end
  end
end
```

require 'dbi'

class TableAdapterFactory
  class Adapter
    Glossary = {
      :pkey_select => "select * from %s where %s = ?",
    }

    def self::lookup( pkey )
      rowdata = nil
      qry = Glossary[ :pkey_select ] % [ table, primaryKey ]
      self.dbh.prepare( qry ) { |sth|
        sth.execute( pkey ) do |sth|
          rowdata = sth.fetch.to_h
        end
      }
      new( rowdata )
    end
  end
end

[...]
def initialize( rowdata )
    @rowdata = rowdata
end

def method_missing( sym, *args )
    super unless @rowdata.key?( sym.to_s )
    meth = %{def #{sym}; @rowdata['#{sym}']; end}
    self.class.module_eval meth, __FILE__, __LINE__
    self.send( sym, *args )
end
end # class Adapter

[...]
def initialize( rowdata )
  @rowdata = rowdata
end

def method_missing( sym, *args )
  super unless @rowdata.key?( sym.to_s )
  meth = %{def #{sym}; @rowdata['#{sym}']; end}
  self.class.module_eval meth, __FILE__, __LINE__
  self.send( sym, *args )
end

data-driven classes in Ruby

Dynamic Factories - A Table Adapter Factory

[...]

[...]
Dynamic Factories - A Table Adapter Factory

```ruby
[...,]
def initialize( *args )
    @dbh = DBI::connect( *args )
end

def getTableClass( table )
    dbh = @dbh
    pkey = @dbh.columns( table ).find { |col| col.primary? }.name
    Class::new( Adapter ) {
        @dbh = dbh; @table = table; @primaryKey = pkey
        class << self
            attr_reader :dbh, :table, :primaryKey
            end
        }
    end
end # class TableAdapterFactory
```
Dynamic Factories - A Table Adapter Factory

```ruby
def initialize( *args )
  @dbh = DBI::connect( *args )
end

def getTableClass( table )
  dbh = @dbh
  pkey = @dbh.columns( table ).find { |col| col.primary? }.name
  Class.new( Adapter ) {
    @dbh = dbh; @table = table; @primaryKey = pkey
    class << self
      attr_reader :dbh, :table, :primaryKey
      end
    }
  end
end # class TableAdapterFactory
```
Dynamic Factories - A Table Adapter Factory

```ruby
[...]  
  def initialize( *args )  
    @dbh = DBI::connect( *args )  
  end

  def getTableClass( table )
    dbh = @dbh
    pkey = @dbh.columns( table ).find { |col| col.primary? }.name
    Class::new( Adapter ) {  
      @dbh = dbh; @table = table; @primaryKey = pkey
      class << self
        attr_reader :dbh, :table, :primaryKey
        end
    }
  end
end # class TableAdapterFactory
```
Dynamic Factories - A Table Adapter Factory

```ruby
[...]
def initialize( *args )
    @dbh = DBI::connect( *args )
end

def getTableClass( table )
    dbh = @dbh
    pkey = @dbh.columns( table ).find { |col| col.primary? }.name
    Class.new( Adapter ) {
        @dbh = dbh; @table = table; @primaryKey = pkey
        class << self
            attr_reader :dbh, :table, :primaryKey
            end
    }
end
end # class TableAdapterFactory
```
Dynamic Factories - A Table Adapter Factory

```ruby
[...]
  def initialize( *args )
    @dbh = DBI::connect( *args )
  end

  def getTableClass( table )
    dbh = @dbh
    pkey = @dbh.columns( table ).
      find { |col| col.primary? }.name
    Class::new( Adapter ) {
      @dbh = dbh; @table = table; @primaryKey = pkey
    class << self
      attr_reader :dbh, :table, :primaryKey
    end
    }
  end
end # class TableAdapterFactory
```
Dynamic Factories - A Table Adapter Factory

```ruby
# Mysql example schema:
CREATE TABLE user (  
  rowid int(11) auto_increment primary key,  
  username varchar(16) NOT NULL default '',  
  realname varchar(250) default NULL,  
); 
INSERT INTO user VALUES  
(1,'zerocool','Dade Murphy'),  
(2,'acidburn','Kate Libby');

factory = TableAdapterFactory::new \  
  'dbi:Mysql:adapter-example', 'username', 'password'  
User = factory.getTableClass 'user'

pr = User::lookup( 1 )  
puts "Protagonist: %s (%s)" % [ pr.realname, pr.username ]  
li = User::lookup( 2 )  
puts "Love interest: %s (%s)" % [ li.realname, li.username ]
```
Dynamic Factories - A Table Adapter Factory

galendril pts/9 [~/source/ruby/OSCON2003]
108/571 {0}$ ruby code/dynfactory.rb
Protagonist: Dade Murphy (zerocool)
Love interest: Kate Libby (acidburn)
Metaclasses

• The highest level of abstraction
• All parts of the class are data – no source on disk
• Of limited practical use, but can be used to generate entire class hierarchies programmatically.
• Makes writing a test suite much easier, as it can be generated from the same data
• May result in a visit from the OOP police.
Consequences: The good, the bad, and the angry DBA.
The good

- Minimize the impact of changes to the code caused by changing data requirements
- Is one of the easiest kinds of code to reuse
- Can condense cut-and-paste reuse into small, easily-maintained chunks.
The bad

- Overuse can make a design overly general
- Can be tricky to test
- Requires more discipline on the part of developers
- Only solves half of the problem – changes are still necessary in the application code when the library changes.
Thank you for listening.

For sample code, a PDF of the presentation, and more:
http://www.rubycrafters.com/