Convention over Configuration

The Universal Remote: Powerful, but requires too much configuring

Intent

Design a framework so that it enforces standard naming conventions for mapping classes to resources or events. A programmer only needs to write the mapping configurations when the naming convention fails.

Motivation

General-purpose frameworks usually require one or more configuration files in order to set up the framework. A configuration file provides a mapping between a class and a resource (a database) or an event (a URL request). As the size and complexity of applications grow, so do the configuration files, making them harder to maintain.

Below is an example[1] showing a typical configuration file.

```
<hibernate-mapping>
<class name="User" table="users">
  <id name="ID" column="id" type="string">
      <generator class="assigned"></generator>
  </id>
  <property name="password"  column="password"  type="string" />
</class>
</hibernate-mapping>
```

Figure 1: A Hibernate mapping definition

```
CREATE TABLE users (
  id VARCHAR(20) NOT NULL,
  password VARCHAR(20),
  PRIMARY KEY(id)
);
```

Figure 2: The `Users` table in the database

Figure 1 above shows a mapping file for Hibernate[5], an object/relational persistence and query service framework for Java. The Hibernate mapping definition in Figure 1 maps the `User` class to the `Users` table in the database. The `Users` table in the database is described in Figure 2 using SQL. The fields of class `User` are also mapped to the columns in the `Users` table.

Hibernate uses the configuration file in Figure 1 to map objects to the database. For instance, if we create an object `Alice` of type `User`, calling `Alice.getId()` will actually perform a lookup to the relevant row in the `Users` database table and retrieve the information in the `id` column.

Modifying configuration files, usually in XML, is tedious and also error-prone. Most errors in configuration files are only detected at runtime in the form of a runtime error, usually a `ClassNotFoundException`. More importantly, a lot of the mapping information can be inferred easily from the structure of the database table without the need for any configuration.

For instance, we can set up the naming convention that:

1. Database table names should be the pluralized form of the class name.
2. The columns in a database table should have identical names with the fields in the class that it maps to.

These two naming conventions are *natural*. As can be seen above, the configuration file is indeed echoing the convention. In fact, most developers already adhere to certain naming conventions when they program. The Convention over Configuration pattern rewards developers for adhering to naming conventions and enforces this in a stricter manner by building it into the framework.

When the designer of a framework establishes a standard naming convention, there is little need for the configuration file. The framework has code that will invoke the relevant classes or methods based on their names. In this example, the framework queries the table for the table name and the names of the fields when a method is invoked. And if the corresponding class does not have a corresponding field for a column name, when we access it for the first time we get a runtime error. This is no different from getting a runtime error in the event that we specified the configuration file wrongly.

The Convention over Configuration pattern reduces the amount of configuration by establishing a set of naming conventions that developers follow.

Applicability

As the designer of a framework, use the Convention over Configuration pattern when

- there are clear and practical naming conventions that can be established between parts of the framework.
- there is the opportunity to reduce the amount of configuration files that *duplicate* mapping information from other parts of the system.

The Convention over Configuration pattern does not preclude the need for configuration files. Configuration files are still important for the cases where convention fails. But for most cases, sticking to the conventions works and keeps things simple for the programmer and anyone reading the code.

Whenever a configuration property is set explicitly, that property overrides the underlying naming convention. Thus the framework is still fully configurable.

The configurations can be specified in a separate file (shown in Figure 1) or the configurations may be embodied in the code, as we shall in the *Sample Code* section.

Consequences

The Convention over Configuration pattern has the following benefits and liabilities:

1. + *Allows new developers to learn a system quickly*. Once developers understand the naming convention, they can quickly start developing without worrying about writing the configurations to make things work. This gives developers the impression that the framework Works Out of the Box[15] with little or no configuration. Frameworks that work out of the box empowers developers to quickly create prototypes for testing. Compare this to frameworks that require multiple configuration files to get the system up and running even for simple tasks. After they have become more familiar with the framework, they can write configurations for the unconventional cases.

2. + *Promotes uniformity*. Developers working on different projects but using the same framework can quickly grasp how different systems work since the same naming conventions are promoted throughout the framework. This helps in maintaining a ubiquitous language[3] for the development team.

3. + *Better dynamism*. Changing the name of the class or method in the source code does not require modifying a configuration file. Since the framework does not rely on static configuration files, but rather enforces the naming conventions during runtime, changes made are automatically propagated through the application.

> *"This is the problem with conventions – they have to be continually resold to each developer. If the developer has not learned the convention, or does not agree with it, then the convention will be violated. And one violation can compromise the whole structure." -Robert C. Martin[2]"*

4. - *Requires familiarity*. The naming conventions become part of the implicit knowledge of the framework. Once a set of conventions has been established, it becomes hard to change them. In fact, not following those conventions makes the system harder to use. Naming conventions have to be included in the documentation and followed consistently in code samples to avoid confusion.

5. - *Larger framework*. By shifting the responsibility of configuration from the developer, the framework itself has to enforce those conventions; the set of conventions has to be baked into the framework. If there are a large number of conventions that need to be supported, the framework becomes larger. Thus, only enforce clear and practical naming conventions in the framework itself.

6. - *Hard to refactor existing frameworks to adopt a new naming convention*. It might not be feasible to use Convention over Configuration when an existing framework has a large group of developers using it. There are currently no automated tools that can upgrade an application to use features in a newer version of the framework. So developers using a version of the framework that used an older convention cannot upgrade easily to a newer convention. The Convention over Configuration pattern is best used during the initial creation of the framework and maintained throughout updates to the framework.

Usage

The naming convention should be distilled from the ubiquitous language[3] if one exists. The naming convention can also be distilled from existing applications if the framework has been designed using the Three Examples[14] pattern.

As the framework evolves, existing naming conventions may have to be modified or new naming conventions have to be added. This set of naming conventions has to be promoted to all the developers that are using the framework. Then, as the designers of the framework, enforce this set of conventions as part of the framework. Enforcing the naming convention is done using the reflection and metaprogramming properties of the programming language.

Also, design the framework to accommodate for the ability for developers to configure if the conventions do not fit.

Sample code

Suppose that we wanted to define a simple validator for some of our classes. We will make use of Convention over Configuration to do this. First, we define a set of naming conventions. If our class is called `Cat`, the validator will be called `CatsValidator`. We pluralize the class name and append the word "Validator" to it. Second, in the CatsValidator class, methods for validating are called `valid_x` where x can be any word. Those are our naming conventions.

We will be using the Ruby programming language to define the classes since it supports reflection and metaprogramming with a clean syntax.

```
class NamesValidator

  # Checks that first_name and last_name are within certain length
  def self.valid_length?(name)
    name.first_name.length < 20 and name.last_name.length < 10
  end

  # Checks that first_name and last_name have the first character capitalized
  # capitalize turns HELLO into Hello; hello into Hello; etc
  def self.valid_case?(name)
    name.first_name == name.first_name.capitalize and
    name.last_name == name.last_name.capitalize
  end

  def self.non_conforming_method
    # This method will not be called during validation
  end

end

class Name < Validatable

  attr_accessor :first_name, :last_name # create getters and setters for instanc
e variable name

  def initialize(first_name, last_name)
    @first_name, @last_name = first_name, last_name
  end

end
```

Figure 3: Name `class` and the NamesValidator `class`

`Name` is just a simple class that has two fields: `first_name` and `last_name`. The `NamesValidator` has two class methods that check if the name is of valid length and if it has the right case. The method `non_conforming_method` is left there to show that our validation system does not call that

method since it does not conform to the naming convention we agreed upon.

```
class Validatable

  # Uses metaprogramming to construct a new validator
  def validate
    unless @validator
      validator = instance_eval(self.class + 's' + "Validator")
      validator.methods.grep /^valid_/ do |m| # find methods like valid_x
        # call the method and passes in this object as an argument
        # puts an error if that method did not validate
        puts "Method " + m + " failed" unless validator.send(m, self)
      end
    else
      validator = instance_eval(@validator)
      validator.methods.grep /^valid_/ do |m|
        puts "Method " + m + " failed" unless validator.send(m, self)
      end
    end
  end

end
```

Figure 4: The base class that enforces the Convention over Configuration

This is the conventional case: if the `validator` instance variable is not given an explicit value (by default, it is nil), the `validate` method constructs a new `Validator` based on the name of the current class. It then searches for methods that have the name `valid_x` and calls them using the `send` method. If the validation fails, then a message telling the name of the failing method will be printed.

```
name = Name.new("Nicholas", "Chen")
name.validate # produces nothing

name = Name.new("Nicholas", "chen")
name.validate # produces Method valid_case? failed
```

Figure 5: Result of validating two `Name` objects

This is the configuration case: if the `validator` instance variable is defined, then the `validate` method uses that for the class name of the `Validator`.

```
class GroceriesValidator

  # Checks the weight
  def self.valid_weight?(grocery)
    grocery.weight < 50
  end

  # Checks the size
  def self.valid_size?(grocery)
    grocery.size < 10
  end

  def some_other_method
    # Not used in this case
  end

end

class Grocery < Validatable
```

```
  attr_accessor :weight, :size

  def initialize(weight, size)
    @weight, @size = weight, size
    @validator = "GroceriesValidator"  # The validator is defined here
  end

end
```

Figure 6: Grocery class and the GroceriesValidator

Because the plural of the word “grocery” is “groceries” our simple convention cannot take care of it. So we have to explicitly specify a mapping by giving the instance variable `validator` the valid Validator class name.

```
grocery = Grocery.new(30,20)
grocery.validate #produces Method valid_size? failed
```

Figure 7: Result of validating an invalid `Grocery` object

Known Uses

Ruby on Rails. One of the driving forces behind the Convention over Configuration pattern is the Ruby on Rails[10] framework. Ruby on Rails, or just Rails, is an open source web application framework written in Ruby that closely follows the Model-View-Controller (MVC) architecture. The concept of minimal configuration has been built-in to the framework.

For instance, Rails provides its own ActiveRecord[7] library that deals with mappings between a class and a table in the database. By convention, the table name is a pluralized form of the class name. Thus, the class `Post` will have a table called `Posts`. Rails also handles peculiar words such as sheep which has the plural form sheep and octopus which has the plural form octopi. For more complex or legacy table names such as `ShopInventory`, the user would have to configure the mapping between the table name and class name.

The database mapping convention is just one facet of the Convention over Configuration pattern in Rails. The pattern is also prevalent in the URL dispatcher in Rails. For further information, readers should consult the documentation for the Rails framework[10].

Spring MVC framework. The URL Mapper in Spring[12] uses the Convention over Configuration pattern to map URLS to the correct `Controller`. A `Controller` object in the Spring framework follows the Page Controller[9] pattern and is in charge of handling a request for a specific page or action on a web site. For instance, a request for *mywebsite.com/accounts* will automatically call the `AccountsController` object. The underlying convention[13] is to use the word after the main URL (accounts), capitalize it, and append the word “Controller” to it. The resulting name, `AccountsController`, is the name of the class that we will forward the URL request to.

Unit testing. The JUnit[6] testing framework (part of the XUnit testing frameworks) by Erich Gamma and Kent Beck make use of conventions to simplify the creation of unit tests. Each method that the developer wants to test is prepended with the word `test`. The framework would then use the reflection properties of the language to locate those methods and execute them. Moreover, the methods `setUp` and `tearDown` are methods that are automatically run before and after each test method. The names of the functions in the XUnit frameworks are not configured; they are by convention.

Related works

Convention over Configuration upholds the Don’t Repeat Yourself[4] philosophy. When the convention is clear and practical, there is little need to duplicate the information. Duplicating the information makes changes hard because the developer has to go to multiple configuration files to make the changes. Martin Fowler identifies duplicated code as one of the infamous code smells[11].

Metadata Mapping[8] using reflection is an example of Convention over Configuration for accessing database tables with objects.

Discussion

A viable *addition* to Convention over Configuration is using an Attribute-Oriented Programming tool such as XDoclet[16]. XDoclet tags can be embedded as metadata before each class declaration in the source code. When the XDoclet tool is initialized, it goes through the source code and gleans the XDoclet tags for metadata. Using that information, XML configurations files are auto-generated to suit the framework.

The advantage is that there is no need to maintain separate XML files for the configuration. The important metadata is embedded in the code and then extracted to auto-generate the configuration files. Therefore, the developer has less files to keep track of and never needs to deal directly with the configuration files.

Even then, XDoclet tags have to be updated manually as well each time the class or method is refactored (rename or moved). So developers can still end up with inconsistent or invalid configurations when they use XDoclet to auto-generate the configuration files.

Therefore, use Attribute-Oriented Programming in addition to Convention over Configuration. When the conventions fail and configurations have to be written, embed those configurations in the code and use a tool to extract that information and auto-generate the configuration files. That way the developers have less files to maintain.

References

[1] Better, Lighter, Faster Java http://www.oreilly.com/catalog/bfljava/

[2] "Convention over Configuration" http://www.dehora.net/journal/2005/11/convention_over_configuration.html

[3] Domain-Driven Design, http://domaindrivendesign.org/book/

[4] "Don't Repeat Yourself" http://en.wikipedia.org/wiki/Don't_repeat_yourself

[5] "Hibernate" http://www.hibernate.org/

[6] "JUnit" http://junit.sourceforge.net/

[7] "P of EAA: Active Record" http://www.martinfowler.com/eaaCatalog/activeRecord.html

[8] "P of EAA: Metadata Mapping" http://www.martinfowler.com/eaaCatalog/metadataMapping.html

[9] "P of EAA: Page Controller" http://www.martinfowler.com/eaaCatalog/pageController.html

[10] "Ruby on Rails" http://www.rubyonrails.org/

[11] "Smells To Refactoring" http://wiki.java.net/bin/view/People/SmellsToRefactorings

[12] "Spring Framework" http://www.springframework.org/

[13] "The Spring Framework – Reference Documentation" http://static.springframework.org/spring/docs/2.0.x/reference/mvc.html#mvc-coc

[14] "Three Examples" http://st-www.cs.uiuc.edu/~droberts/evolve.html

[15] "Works Out of the Box" http://www.laputan.org/selfish/selfish.html#WorksOutOfTheBox

[16] "XDoclet" http://xdoclet.sourceforge.net/xdoclet/index.html

Written by Nicholas Chen. Last significant update on November 29, 2006