Identifying Anti-Patterns in Ruby on Rails Using Automated Detection of Code Smells

Robert O'Regan

A thesis submitted in part fulfillment of the degree of

MSc in Advanced Software Engineering

Supervisor: Dr Mel Ó Cinnéide

UCD School of Computer Science and Informatics
College of Engineering Mathematical and Physical Sciences
University College Dublin

April 2011

Abstract

Refactoring is a software engineering technique that, by applying a series of small behaviour-preserving transformations, can improve a software system's design, readability and extensibility. Code Smells are indications that a piece of code could benefit from refactoring. In this work we focus on refactoring in the context of Software Frameworks.

The goal of this work was to develop a Code Smell detection system for use with the Ruby on Rails framework. Unlike most common smell detection systems which try to examine the catalogue of "classic" code smells, here we take a slightly different approach. We extend the concept of a Code Smell to encompass common bad practices that are often encountered when developing using the Ruby on Rails Framework.

Trials of the detection system are performed on a well known open-source Rails application and the results are evaluated by a number of experienced Rails developers.

Acknowledgments & Dedication

I would like to thank my supervisor, Dr Mel Ó Cinnéide, for inspiring this work and guiding it through his patience, advice, encouragement and constructive criticism.
Thanks also to my family and friends for their patience and encouragement, not only during this work but over the last two years.
I would especially like to thank Sheena who on top of being an endless source of encouragement and support, undertook the task of proof reading this work.
And finally I would like to mention my wonderful daughter Evie who was always a very welcome distraction.

This work is dedicated to my late father Bobby O'Regan.

Table of Contents

1. Introduction

This work forms part of a thesis project completed as part of a Masters in Advanced Software Engineering at University College Dublin, Ireland. The goal was to build and evaluate an automated *Refactoring* tool for use with the Ruby on Rails [1] framework.

Refactoring is the process of analysing code in an effort to identify areas where it could be made clearer, cleaner, simpler and possibly more efficient. Although refactoring had existed in various forms for many years, William Opdyke's 1992 thesis, *Refactoring Object Oriented Frameworks* [2], was one of the first papers to specifically examine refactoring. Martin Fowler summed it up succinctly as…

> "The process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" [3]

Refactoring has become a popular topic and is the subject of a vast number of articles, papers and books. It is usually done in a number of small behaviour preserving steps. After each step you are left with a working system that functionally hasn't changed. Refactoring implies the mathematical term of equivalence in that the beginning and end products must be functionally identical.

There are various forms of refactoring. A common refactoring process is to improve the design of existing code to take advantage of Design Patterns [4]. The use of common Design Patterns enhances a developers ability to read and extend that code. Another form of refactoring is the identification of what are termed as *Code Smells [3]* in a code-base and their subsequent removal. A Code Smell is a metaphor used to describe possible anti-patterns that are generally associated with bad design and possibly bad programming practices.

In general, refactoring improves the readability, extensibility and maintainability of code. Code that is easier to read and understand is usually easier to maintain.

1.1 Thesis Aim

In this work we will build a prototype code smell detection system for use with the Ruby on Rails framework. Unlike most common smell detection systems which try to identify the "classic" catalogue of code smells, here we take a slightly different approach and extend the concept of a code smell to encompass the common bad practices that are often encountered when developing applications using Web Frameworks. The detector aims to provide feedback about the quality of an existing application and also of those under development.

In this work we describe the project itself, give an overview of related work and examine the background and motivation. In particular we focus on the increasingly popular Framework known as Ruby on Rails which utilises a Model-View-Controller (MVC) [5] architecture. We will explore the concepts of code smells, automated smell detection and static code analysis and look at how they can be applied to identify common bad practices developers employ when constructing applications using web application framework. We detail the development process of the detector, present the code smell detection process that was followed and evaluate the effectiveness of the detector by running it on a well known open-source Rails application.

1.2 Thesis Layout

This section provides a brief overview of the remainder of this report.

Chapter 2 details the background and motivation behind this work, discussing some of the technologies and concepts involved and examining possible solutions.

Chapter 3 looks at the design and development of the smell detection tool. This tool forms part of this project and is used to prove some of this works concepts.

Chapter 4 examines some of the common bad practices that are encountered in Rails applications and proposes methods of detection that will be implemented in the smell detector.

Chapter 5 discusses the results of an evaluation of the tool by testing it on a well known open-source Rails application and analyses its results among a group of experienced Rails developers.

Chapter 6 states the conclusions drawn from this works and proposes some possible future enhancements.

2. Background & Motivation

This chapter aims to put this work into context. In Section 2.1 we detail Software Frameworks, Section 2.1 discusses Web Frameworks, while Section 2.3 looks at the common architectural pattern known as Model-View-Controller which is the architecture adopted by most Web Frameworks. In Section 2.4 we take a look at the Ruby on Rails framework upon which this project is based. Section 2.5 then goes on to discuss the motivations behind this project and how we aim to resolve the issues identified.

2.1 The Advent of Software Frameworks

There are many facets to software development, ranging from development methodologies such as Waterfall, Agile and Scrum to the more abstract concepts embodied by software architecture as a whole.

The concept of software architecture was first identified in the late 1960's and early 1970's by Edsger Dijkstra and David Parnas. Their research examined the importance of structure in software systems. While software architecture has always been an integral part of computer programming, the 1990's in particular saw a concerted effort to improve the structure and better define its fundamental aspects. From this grew many concepts such as Patterns (architectural and design) and software frameworks.

Software Frameworks provide developers with tools to develop more flexible and less error-prone applications. They can be thought of as a type of architectural pattern, containing all the necessary elements to quickly and efficiently construct a software application. They are used to define an architecture that an application is constructed with. Often referred to as Architectural Frameworks, they have seen an explosion in popularity in the last decade.

Frameworks allow developers to spend less time developing and debugging and more time on the business specific problem by alleviating the burden of dealing with the "plumbing" of an application. They define an applications global architecture, allowing developers to extend and enhance the frameworks functionality at specific extension points. Essentially they define the flow of control for the application being developed.

Frameworks encourage code re-use, which helps to reduce the development and testing effort and increases code reliability. They can help to establish better programming practices through the use of well established concepts such as design patterns.

However, in spite of the obvious advantages, utilising frameworks as they were intended can be error-prone. Often developers who are new to a particular framework will carry old habits with them. They may be used to the nuances of a previous framework and find it hard to let go of its conventions. A solution may be implemented that suits a previous framework but is at odds with the current framework. It is also tempting for developers to misplace items such as business logic simply out of handiness.

Frameworks provide the scaffolding on which an application can be built. It is up to the development team to flesh out this scaffolding by adding business logic and data. It is this process that can often lead to issues as developers may stray from the intended structure as they race to add functionality to the application.

2.2 Web Application Frameworks

By design, the World Wide Web is not inherently dynamic. In the early days of the web, static Hand-Coded HTML pages were common. The Common Gateway Interface (CGI) was an early attempt at providing a more dynamic way to access and construct web resources. This was soon followed by web servers which supported the creation and addition of modules enabling it to generate dynamic content through the use of technologies such as Java. This was soon followed by a wave of increasingly popular technologies known as scripting languages, led by the likes of Perl, Python and PHP. [6]

While these technologies provided solutions for the creation of dynamic web content, they all still had issues when it came to efficiently creating a web based application. Developers found themselves encountering the same issues when building new web applications. Troublesome issues like handling database connections and authenticating users were common requirements of web based applications. To solve some of these problems, developers would find themselves copying code from one project to another or worse still reinventing the wheel and building the same code from scratch. These inefficiencies in how web based applications were created was eventually aided with the concept of web application frameworks.

A Web Application Framework is a type of software framework, providing a developer with the means to efficiently develop web based applications. They offer libraries to handle things like database access, user authentication, session management and they each implement a specific architectural style.

Web application frameworks have exploded in popularity over the last few years. Rails[1], JavaEE [7], CakePHP [8], Django [9] and Spring [10] are all examples of popular web application frameworks. Each framework implements a particular architectural style such as Model-View-Controller (MVC) [5], Model-View-Presenter (MVP) [11] and N-Tier [12].

Most web application frameworks are based on the Model-View-Controller architectural pattern which is the architecture that this work will focus on.

2.3 The Model–View–Controller Architecture

MVC is a classic architectural pattern that was first described by Trygve Reenskaug [13] in 1979 while working on Smalltalk at Xerox PARC. It isolates application logic from the presentation by the clean separation of an application into distinct layers. This layering of an applications distinct features ties in nicely with the concept of 'Separation of Concerns' as coined by Dijkstra [15].

MVC is comprised of three layers which are described below and depicted in Figure 2.1.

- The Model
  The Model represents the business data and business logic of the application. One of it's main functions is to provide a wrapper for all interactions with a database. This means reads, writes and updates are typically handled by the Model. It also contains the main application logic in the form of any calculations that might be performed on the applications data. The Model responds to requests for information about its state and responds to instructions that request a change of state.

- The View
  The View is responsible for handling how information is presented to the user. A view typically takes the form of a webpage, although it can also take other forms such as XML or JSON if it is being used as a Web-Service for example.

- The Controller
  The Controller defines application behaviour and serves as an intermediary between the Model and the View. It interprets user inputs and maps them to the actions provided by a Model. Its then selects the appropriate View based on the user inputs and the outcome of Model interactions. Controllers are best at parsing inputs, calling the appropriate models, and then formatting the outputs.

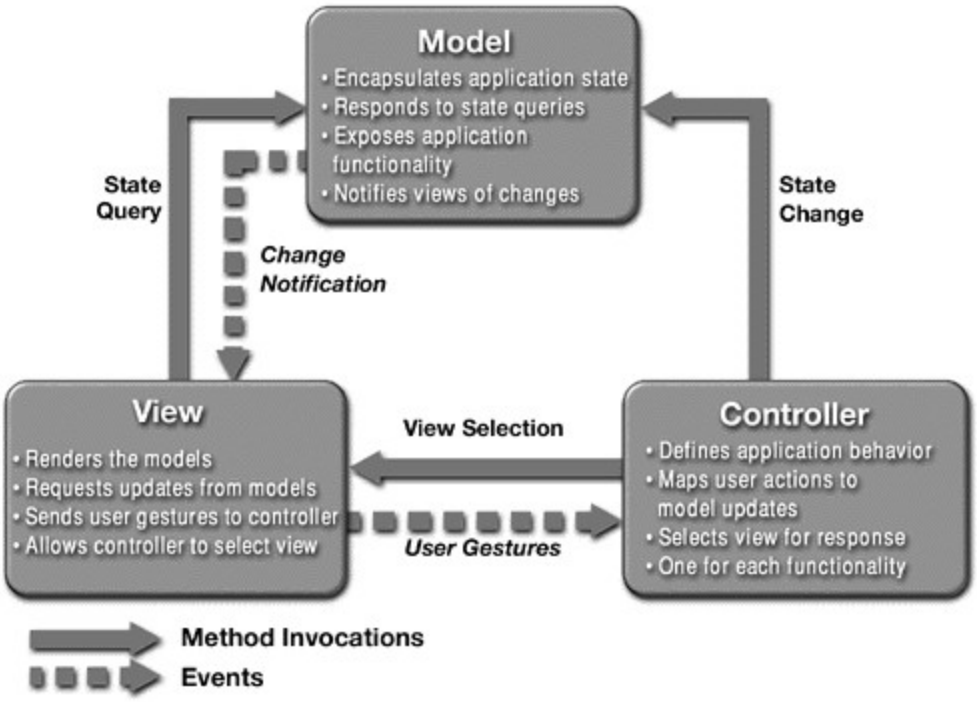


Figure 2.1 – MVC Relationships [16]

Essentially, in an MVC based application, Controllers handle input events from users (such as an HTTP GET request), interpret the event, call the appropriate Model action and select the View to handle output/presentation to the user.

2.4 The Ruby on Rails Framework

Ruby on Rails (RoR) is a Web Application Framework which implements the MVC architectural pattern. It was created by David Heinemeier Hansson in 2004, stemming form work on another project [17]. Rails is a framework which was written in and utilises the Ruby language. Ruby is a fully object-oriented, dynamically typed, programming language that was first developed by Yukihiro Matsumoto in the mid 1990's [18].

Ruby on Rails is designed to make developing web applications easier by making some key assumptions about how web applications "should" be built. It emphasises the design paradigm of 'Convention over Configuration' (CoC) [19] in how it maps its Models to relational database tables and encourages the principle of 'Don't Repeat Yourself' (DRY) as coined by Andy Hunt and Dave Thomas in their book 'The Pragmatic Programmer' [20]. DRY essentially means that repeating the same code over and over is a bad thing and should be avoided as much as possible.

Part of the reason for Ruby on Rails success is down to the basic tools that are provided as standard. Code generation is available in the form of “scaffolding” which will build the outline of the any specified Models, Views and Controllers. Ruby programs, extensions and libraries can be easily distributed and installed using the RubyGems package manager and Rake is Rubys equivalent of Make. Rails also supports Web Services as standard. Responses can be formatted in HTML or XML and support for other formats such as JSON is available through the package manager. These tools provide a stable and well supported development environment.

The Rails framework contains a wealth of packages which are responsible for all the common “plumbing“ requirements that are associated with Frameworks. ActiveRecord maps Models to Relational Databases, ActiveResource provides web services, ActionPack is responsible for handling web requests and ActionMailer enables emails to be sent from an application.

Ruby on Rails implementation of MVC is slightly different to how other frameworks implement the pattern. In Rails, the controller issues instructions to the View to render itself, providing it with access to data in the Model. In other frameworks implementing MVC, the view can interact with the Model directly or is automatically notified by the Model of changes in state (using the Observer Pattern) that require a screen update.

2.5 Web Services

A Web Service is “a software system designed to support interoperable machine-to-machine interaction over a network” [21]. The two applications involved in a web service dialogue may have nothing in common. They may be located on different operating systems and implemented in different technologies.
There are two main classes of Web Services; arbitrary web services which are most commonly implemented as SOAP based services and REST-compliant web services.

2.5.1 SOAP Web Services

SOAP is an XML based protocol that allows applications to communicate and exchange information over HTTP. It defines a set of rules for structuring messages that can be used for simple one-way messaging but is particularly useful for performing Remote Procedure Call (RPC) style request-response dialogues.

Unlike RESTful approaches, SOAP can have a large number of differing services that it offers. These are defined in a Web Services Description Language (WSDL) file. A WSDL file is an XML based language that provides a model for describing web services. Typically, a client application will consume a SOAP web

services WSDL file to gather information on what services are provided. The WSDL file describes the service and how to access it by specifying, for example, what parameters it expects to be sent. The client application can then construct a request in the correct format before submitting it.

While SOAP made web service development accessible and popular, critics often point out that it can be overly complex. A lot of new web services are being implemented using REST instead of SOAP due to it being lightweight and relatively quick and easy to implement.

2.5.2 RESTful Web Services

Representation State Transfer (REST) is an approach to building web applications that takes as much advantage of the underlying structure of the web as possible and as such can be a very neat fit with MVC. REST doesn't create any new techniques. Instead it encourages developers to use old techniques as they were designed to be used.

Web developers have traditionally used two HTTP methods to interact with web applications: GET and POST. GET is generally used to query for data whereas POST is used to update or add data. This is a very limited use of the HTTP protocols suite and the over-reliance on these two methods has led to a number of issues.

For GET requests, in some cases URLs became large making them unreadable for browser users. Worse still, in some applications developers used GET to requests changes to data. GET requests were designed to be idempotent in that each identical GET request should return an identical result.

For POST the problem was much simpler. POST was mainly tasked with doing all the heavy lifting of an application. It would be responsible for handling all requests that modified data. Because of this practically nothing created with a POST requests was book-markable.

HTTP has nine methods defined for use when interacting with a resource located on the web such as a web application. These include HEAD, GET, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT and PATCH. Combining PUT and DELETE with GET and POST gives HTTP a complete set of verbs. Using these allows manipulation of data in a similar way to the well known model referred to as CRUD, which is used in many aspects of software engineering.

CRUD stands for Create, Read, Update and Delete and was coined by James Martin in a 1983 book titled “Managing the Database Environment” [40]. CRUD lends itself in particular to the manipulation of data in relational databases. This basic set of verbs manages practically everything we do with databases. HTTP

provides the extra verbs to use this model when interacting with resources on the web.

As mentioned, this fits very neatly with MVC and in particular Controllers. Controllers handle input from users and delegate the request to the Models and Views. Since practically all data manipulation can be achieved using the CRUD model, we can map the principles of CRUD directly to Controllers. Each Controller will implement the same methods which correspond directly to a particular HTTP verb. In this way Controllers become standardised modules which connect a data model to the Web, instead of modules containing a wide variety of differing actions. This is in contrast to SOAP based web services which use a WSDL file to describe the services that are provided.

With a RESTful Controller the INDEX method answers GET requests for a listing of all available data. SHOW answers GET requests to display a single record. NEW answers GET requests for a form to create a new record but it doesn't actually create a record directly. EDIT answers GET requests for an editable version of a single record. The CREATE method answers POSTs that send new data to create a new record. UPDATE responds to PUTS that send data modifying an already existing record and DESTROY responds to DELETES which remove the requested record.

This is exactly how REST expects the Web to work. URL's connect users to resources on the Web. These resources can be thought of as *nouns* that the HTTP verbs (implemented in Controllers) work on. In essence users are provided with a standardised way of accessing resources within a web application.

In the context of this work we will further examine RESTful methods and how they can help resolve some of the issues we identify. Ruby on Rails already encourages building web applications using a RESTful approach. If you use “scaffolding” to create your controllers, the generated Controller code will already contain all the RESTful methods leaving the developer to fill in the blanks. However adhering to this approach can be where the problems arise. Developing in a RESTful manner requires a shift in the way many developers think about Controllers and web applications in general.

2.6 Anti-Patterns & Code Smells

2.6.1 Anti-Patterns

Anti-Patterns are negative solutions that present more problems than they address. They were first described by Andrew Koenig in 1995 [22] and were inspired by the Gang of Four’s (GoF) book ‘Design Patterns’ [4]. The term gained widespread popularity with the publication of ‘AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis’ [23] in 1998.

Anti-Patterns identify and categorise some of the common mistakes that occur in software development. The term can also be used to describe an individual Code Smell or a collection of smells which point at a larger issue. Anti-Patterns are defined as having two key elements [23].

i. Some repeated pattern of action, process or structure that initially appears to be beneficial, but ultimately produces more bad consequences than beneficial results.
ii. A refactored solution exists that is clearly documented, proven in actual practice and repeatable.

In the context of this work we refer to the collection of Code Smells, as outlined in Chapter 4, as being an indicator of the existence of a wider anti-pattern.

2.6.2 Code Smells

A Code Smell is a metaphor used to describe possible anti-patterns that are generally associated with bad design and bad programming practices. It stems from the study of refactoring and was first coined by Kent Beck while contributing to Martin Fowlers book “Refactoring: Improving the Design of Existing Code” [3]. According to Fowler, refactoring is usually motivated by the identification of a Code Smell.

A Code Smell is an indication that a systems design is diverging from best practices and should be considered for refactoring. The code in question might compile fine and function as expected, but how it was implemented, or how it has evolved, could perhaps be improved upon and is often referred to as being “smelly”.

Some common code smells were identified by Beck and Fowler, and occur across many different languages, such as God Classes, Long Methods, Duplicated Code, Feature Envy and Inappropriate Intimacy. These are commonly referred to as “classic” code smells throughout this work.

Bear in mind that a code smell is only an indication that something might be wrong. It is not a definite declaration that it is wrong. Labeling a piece of code as smelly is not an attack on that code; it is a sign that perhaps a closer examination of the implementation is warranted.

Bad-Smells in software are indications of poor design and possible bad programming practices. These patterns can be removed from the software system by using refactoring techniques which improve the readability, maintainability and comprehension of the software system. In this work we take the concept of a Code Smell and extend it to include common bad practices that are encountered when developing applications using Web Frameworks.

2.7 Motivation

The primary motivation behind this work is the identification of a common Anti-Pattern that has evolved within Web Application development, in particular with Ruby on Rails development. In this work we refer to this as the Fat Controller Anti-Pattern. MVC defines a clear separation of layers with each layer tasked with distinct responsibilities. Even with this layering, bad practice can creep into application development when developers take shortcuts or don't fully understand where a certain application element should go. Ideally in an MVC architecture, Controllers should be skinny, Models should be fat and Views should be dumb.

2.7.1 Skinny Controller, Fat Models and Dumb Views

"Skinny Controllers are intention-revealing. Fat Controllers are intention-obscuring" [24].

Within the Rails community, the topic of Skinny Controllers and Fat Models has received much attention over the last few years thanks in part to Jamis Bucks 2006 article on the subject [25].

When first getting started with development in a framework that employs the MVC pattern, it is common for developers to place a lot of the business logic of the application in the View. This is a direct violation of MVC's principles. The View is responsible for handling how information is presented to the user. That is its sole objective. How that information is constructed is of no concern to the View. Views should be *dumb*. They should have no knowledge of what an application does or how it achieves its objective. Excess Ruby code in a View is an indication that the View perhaps knows more than it should.

Placing business logic in the View also blurs the distinct separation of layers that MVC embodies and violates the "Separation of Concerns" principle upon which it is built. MVC has been successful for many reasons such as its "Separation of Concerns", readability, maintainability and modularity.

In attempting to resolve this issue, some Rails developers may be tempted to move the business logic from the View to a Controller as in some ways it is a logical step. However this is also considered bad practice. As mentioned, the role of a Controller is to handle user events that affect the model or view. Its main responsibility is to act as an intermediary between Models and Views.

This leaves the Model. A lot of developers make the common mistake of assuming that Models are primarily concerned with database interaction, believing it to be the Models sole responsibility. By definition Models are an abstraction of some real world process or system. They capture not only the state of a process or system but also how the system works. In essence, Models define what a system does and how it does it. Also, in placing the business logic

of an application in the Model we are observing the OO principle of putting logic where the data it operates on is located. By following this principle we also simplify a number of other aspects of application design. Not only should business logic be kept in the Model; any validation that is required on the application data should be kept there also. By doing this we keep all relevant logic in one place and ensure that the concept of *loose coupling* is observed between all layers.

This all feeds into the concept of Skinny Controllers and Fat Models. This principle essentially suggests that you should put as much of the business logic of the application into an applications Models and the only thing the Controllers should be doing is retrieving data from the Model and passing it to the View. It is a simple yet powerful concept. Maintaining skinny controllers can help to increase code readability, maintainability and modularity, in keeping with the principles of MVC. Slim Controller actions are a sign of good practice, as are Views that are mostly HTML.

This concept is embodied very well in the phrase: "*We need SMART Models, THIN Controllers, and DUMB Views*" [26].

One problem that can occur with this approach is that Models can start to become obese. By constantly moving more logic and validation to the Model they themselves can become unwieldy. This issue is beyond the scope of this thesis, but can be remedied by splitting the Model into smaller Models with discrete functions or by moving some of the logic out to Modules.

In this work we deem non-adherence to the principles discussed above to be an anti-pattern. In identifying this anti-pattern we examine applications for common Code Smells, which can infer the existence of this anti-pattern and in some cases suggest ways to correct it.

2.8 Resolving the Issues

In this section we examine the issues and background that have been discussed above and look at methods of resolution.

2.8.1 Static Code Analysis

In the smell detection tool developed, we use Static Code Analysis to locate possible code smells which can be an indication of the Fat Controller anti-pattern.

Static Code Analysis is a method of software analysis that is performed by examining a piece of code without actually executing it. Examples include source code reading and code reviews and these are usually guided by rules and guidelines that are contained in coding standards. Coding Standards can be

company or application dependent but all generally try to ensure that the code being analysed is reliable, maintainable, testable, portable and readable.

Static analysis is by no means perfect. It can incorrectly flag a section of code in a program which behaves correctly but is in fact fine (false positive).

Static Analysis has increased in popularity in recent years with the development of numerous automated static checkers which enable developers to detect problems earlier. These tools have evolved from basic syntax checkers to tools that can reason about the semantics of code. An excellent example of how far static analysis has come is the Extended Static Checker for Java (ESC/Java 2) [27]. ESC/Java2 uses JML (Java Modeling Language) annotations to check that an application is following the “Design by Contract” paradigm that was made popular by Bertrand Meyer’s Eiffel programming language [28].

2.8.2 Static Analysis in Dynamic Languages

Static Code Analysis is far more mature in statically typed languages such as Java, than it is in dynamically typed languages such as Ruby. This is partially due to the very nature of dynamically typed languages.

With no type safety net in dynamically typed languages, it is easy to make trivial mistakes and can often be extremely hard to find them in large applications. This is part of the reason why solid and comprehensive testing has become such an important topic in dynamic languages. Most large applications written in a language such as Ruby would have large fine-grained test suites which can sometimes take hours to run. This is a fundamental aspect of developing with dynamically typed languages and one of the perceived drawbacks that comes with the flexibility provided by these languages. However, such testing is not necessarily a bad thing and the advances in unit testing architectures, such as the xUnit family, have only led to make unit testing more efficient and accessible.

For a long time there was a dearth of static analysis tools in the Ruby world. The fundamental building block of static analysis in Ruby is the ability to access the Abstract Syntax Tree (AST) of source code. AST’s, which will be discussed in greater detail in Chaper 3, are a tree-like representation of the syntactic structure of a piece of source code. The tree can be traversed and conclusions drawn on its structure which allow us to comment on its characteristics. An initial solution was the *parse_tree* [29] gem, a Ruby extension written in C which facilitated the creation of AST’s. Support for this gem was inconsistent across Ruby versions and it lacked data such as line numbers in the AST which are crucial for static analysis tools that report the location of possible problems. It never the less kick started the idea of static analysis in the Ruby world. Soon after parse_tree, came the *ruby_parser* [29] gem. This parser was built purely in Ruby, produced the same type of AST’s and fixed most of the issues experienced with parse_tree.

2.8.3 Using Automated Static Analysis to Detect Code Smells

Maintaining the design standards defined by any Framework requires considerable time and effort, but can be greatly aided by automated code smell detectors. While there are limits to how much can be achieved with static analysis in dynamically typed languages, there are still a large number of possibilities that are being actively pursued. One such area that has seen a lot of attention is the automatic detection of common code smells.

Detecting code smells was originally a manual process. Developers would spot "smells" in code and refactor the code to remove that smell. This process evolved and culminated, to some extent, in Fowlers book on Refactoring [3], which proposed a number of common smells that occur in a variety of languages.

With a list of common smells, developers set about creating more efficient ways of detecting these smells, leading to the development of automated smell detectors that analyse code-bases and report any smells that may have been identified. These detectors exist in a number of different languages and on the whole identify similar types of smells, with some focusing on more specific elements of the language being analysed.

It should be noted that a smell detectors accuracy might never be 100%. This is due to a number of reasons such as inherent limits on static analysis and the fact that not everyone will agree that a Code Smell is a Code Smell. We also have the issue of false positives. It is not uncommon for an automated smell detector to label a piece of code as *smelly* when in reality it isn't. There are also cases where a piece of code is *smelly* but for various reasons there are no alternative ways of implementing it.

2.8.4 Existing Rails Static Analysis Tools

There are a number of Static Analysis tools already available in the Ruby world. The vast majority of these are related to the refactoring process and code smell detection. The *ruby_parser* gem facilitates the creation of smell detectors in Ruby through its AST generation. There are already a number of code smell detection tools available in Ruby, most of which aim to detect the catalogue of "classic" code smells described by Fowler and Beck [3]. In this section we discuss some of the more popular Ruby tools.

Reek [30] was developed by Kevin Rutherford and is one of the more well known smell detection tools in the Ruby world. Reek currently includes checks for some aspects of Control Couple, Data Clump, Feature Envy, Large Class, Long Method, Long Parameter List, Simulated Polymorphism, Uncommunicative Name and many more. Reek was originally built using the *ParseTree* gem before being rebuilt using *ruby_parser*.

Roodi [31], by Martin Andrews, is similar in structure to reek. Roodi comes with

built-in checks that among other things, ensure methods and modules comply with proper naming conventions and check parameter counts. Other checks include Ruby specific advice such as avoiding for loops in favour of Rubys iterators. In contrast to Reek, Roodi was originally built using Jruby before being rebuilt to use ruby_parser. This highlights how valuable ruby_parser has become to the Ruby refactoring process.

Flay [32], written by Ryan Davis, checks code-bases for duplicates and code that may have been copy/pasted. Flog [33], which was also developed by Ryan Davis, calculates a score for a code-base which depends on various patterns that are considered bad, such as a class that contains a large number of dependencies.

2.8.5 Using Rails RESTful Principles

As mentioned, Rails comes with full support for implementing RESTful principles. The RESTful approach to developing web applications fits very neatly with the concept of Skinny Controllers and Fat Models and offers a potential solution to many of the problems encountered. Figure 2.1 shows a basic RESTful method that would be used in a Controller.

```
# GET /people
# GET /people.xml
def index
  @people = Person.find(:all)

  respond_to do |format|
    format.html             # show.html.erb
    format.xml { render :xml => @person }
  end
end
```

Figure 2.1

What is clear from the code in Figure 2.1 is that the method is “skinny” and it's intentions are obvious. It uses a Model to access the data it requires and instructs a View to present that data.

Another benefit of developing using the RESTful approach is that we get web services for free. The Controller can decide if the data is to be returned using a normal HTML view or and XML view. We can also add in other View types here such as JSON.

2.9 Summary

This section detailed the background and motivation behind this work. To summarise, we are trying to identify a common Anti-Pattern that occurs in Web Application Framework development. This anti-pattern can be diagnosed by the presence of a number of Code Smells.

The following chapters discuss the development of the Code Smell detection tool, detail the Code Smells that are used to identify the Anti-Pattern and evaluate the tools effectiveness.

3. Design & Development

This chapter details the smell detection tool that evolved, focusing on some of the more interesting areas of its design. Section 3.1 describes the various technologies that were used in the construction of the detector. Section 3.2 discusses the application at a high level, detailing the various phases of smell detection, while Section 3.3 contains a more detailed look at how the detector was constructed.

Initially a number of development spikes were undertaken. This was done firstly to prove the validity of the concepts outlined in Chapter 2 and secondly to give the author a better understanding of the technologies required and used during development.

3.1 Technology Overview

This section details the various technologies that were used in the construction of the smell detection tool.

3.1.1 Abstract Syntax Trees

Compilers and interpreters never work directly on the source code. Instead they parse code into formats that are more efficient to deal with, such as Abstract Syntax Trees (AST). For example, the Ruby language is an interpreted language and is converted into such a tree during the interpretation process.

An AST contains a tree-like representation of the abstract syntactic structure of a piece of source code. Each node of the tree denotes a construct occurring in the source code. An AST is considered abstract in the sense that it does not contain every detail of the original source codes syntax. Instead it is more concerned with the structure and logic of the source code. Take the code snippet shown in Figure 3.1.

```
x = a + 10;
while (x > a)
{
   a = a + 1;
}
```

Figure 3.1

Figure 3.2 shows this code snippet when represented as an AST.

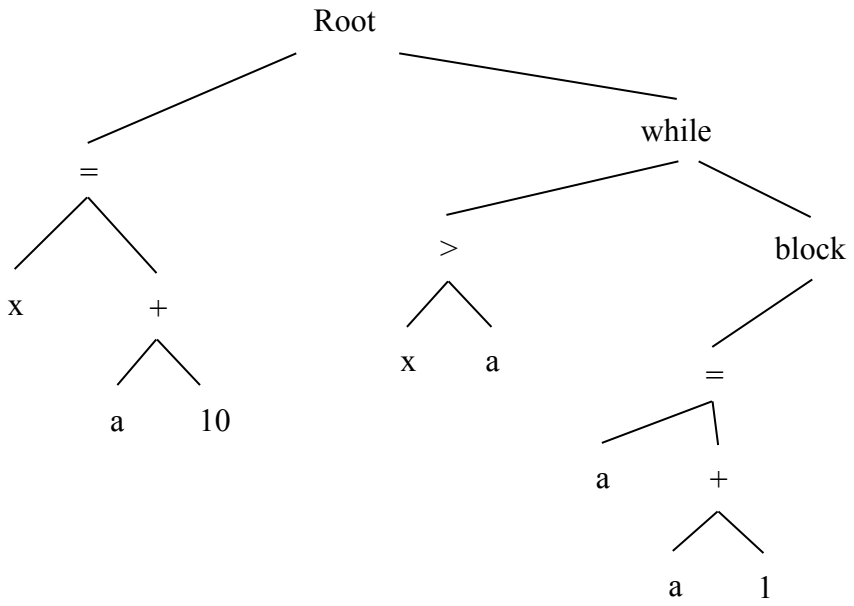


Figure 3.2: Representation of an Abstract Syntax Tree

In the smell detection tool developed as part of this work, we convert the Ruby source code into an AST using a freely available RubyGem known as 'ruby_parser', which is discussed in greater detail in Section 3.1.3.

3.1.2 S-expressions

Abstract Syntax Trees provide an overall structure for representing source code. Each node on the tree structure provides data about the source code itself. These nodes need to be represented in some way. In Ruby we use S-expressions to represent tree nodes.

S-expressions, also known as sexp, are list based data structures for representing complex data. They are a variation on LISP's S-expressions. An S-expression can contain a list of other S-expressions, which in turn may also be lists. So S-Expressions can be arbitrarily nested to any depth and as such are an ideal way of representing Abstract Syntax Trees.

In an S-expression, the first element in the list is the operator/function and the rest are the operands/arguments.

For example, a simple expression is show in Figure 3.3.

```
(2 * (3 + 4))
```

Figure 3.3

Figure 3.4 shows how this same expression would be represented as an S-expression.

```
* 2 + 3 4
```

Figure 3.4

Because AST's represent the abstract structure of source code, we really only need to deal with a small number of node types when constructing such a tree structure. The main node types used are *Def*, *Call* and *Lit*.

Def can be used to describe a method and has the following structure…

```
[Def, method_name, arguments, body]
```

This node type represents a method with the method name, method arguments and body of the method contained in the node. Remember the *body* element of the node can in turn contain nested nodes and so on.

Lit can be used to represent variable assignment and is structured as…

```
[Lit, number]
```

In this node type *number* will be replaced by the value being assigned to the variable.

Call can be used to represent code flow and is structured as…

```
[Call, receiver, method_name, arguments]
```

This node type is used to represent an operation taking place or a method being called.

These three basic node types can be used to describe the structure of most code-bases. The method in Figure 3.5 highlights this.

```
def add_ten(n)
    n + 10
end
```

Figure 3.5

When represented as an S-Expression becomes…

```
[Def,
 add_ten
 [n]
 [Call, n, +, [[Lit, 10]]]]
```

Figure 3.6

S-Expression support is available in Ruby through a gem titled *sexp*, which is packaged as a dependency of ruby_parser.

3.1.3 Ruby Parser

ruby_parser (RP) [29] is a parser written in pure ruby which allows you to quickly build an AST of a specified piece of Ruby source code. It was developed by Ryan Davis at 'seattle.rb' [29] and is the successor to the ParseTree gem which was the first tool that enabled the generation of ASTs in Ruby.
The AST nodes that 'ruby_parser' generates contain S-Expressions which are represented using Ruby's array, string, symbol, and integer types.

Using the sample code from Figure 3.5, the AST that 'ruby_parser' generates would look like Figure 3.7.

```
s(:defn,
 :plus_five,
 s(:args, :n),
 s(:scope,
  s(:block,
   s(:call, s(:lvar, :n), :+, s(:arglist, s(:lit, 5))))))
```

Figure 3.7

3.1.4 Erubis

One issue that was encountered during development was related to creating an AST of Ruby code embedded in Views. The ruby_parser gem expects to be fed clean Ruby code such as a class or method. To "clean" up the View code we need to extract the embedded Ruby code before it is passed to 'ruby_parser'. The Erubis [33] gem enables us to do this.

Erubis is an alternative View renderer for Ruby and is used in the Merb [34] framework as its default renderer. Erubis extracts Ruby code embedded in Views allowing it to be executed. In the smell detection tool we use Erubis to simply extract embedded Ruby from View which we can then pass to 'ruby_parser' to generate an AST.

Erubis will actually become the default renderer from Rails 3 onwards due to its improved performance.

3.1.5 Rubys Open Classes

Also known as Monkey-Patching [35], Ruby, unlike most other Object Oriented languages, is unique in that all its classes are open. This includes its core/library classes like String and Array. In most other languages, once a class has been defined you cannot add any more methods to it. In Ruby however, we can add new methods very easily. This feature makes the language incredibly flexible.

Take the following basic example. Assume that the Ruby String class didn't contain a method to convert a string to lowercase and we needed just such a method. In other languages we might have to create a new utility class containing a method which does the conversion. We would then have to call that class/method, returning the modified string. In Ruby we can simply open the String class, add a new method and call it as a method on that string as follows…

```
Class String
   def to_lower
      # do conversion...
   end
end
```

Then call the new method as we would any other string method as follows…

```
puts my_string.to_lower
```

Using this language feature in the detection tool being developed, we can add an "accept" method to the sexp class which allows us to use the *Visitor Design Pattern* [4] to *visit* each node in the AST.

Monkey-Patching is the subject of much debate amongst software developers. Python developers look down on it while Ruby developers see it as a useful feature. Monkey-Patching has obvious dangers but does allow for increased flexibility in terms of what can be achieved. Indeed, much of the functionality

provided by the Rails framework is achieved by *monkey-patching* some of Rails core classes.

Whether adding a new method to a base class, that a developer finds useful, is good practice or not is beyond the scope of this document. However in the context of this smell detector, the additions we make to the *sexp* class are fairly minimal and in this case Monkey-Patching serves as a useful feature.

3.2 High Level Design

The detection of code smells is achieved using a number of steps as outlined in Figure 3.8.

A parser analyses the Ruby source code files (Models, Views & Controllers) and produces Abstract Syntax Trees (AST), which contain all the relevant structural information. The AST enables us to perform a static analysis of the code. Once we have constructed the AST, a Visitor traverses the tree, and executes *smell checks* against the nodes it is currently visiting. If a smell is detected in a node an error is generated, highlighting that node as being potentially smelly.

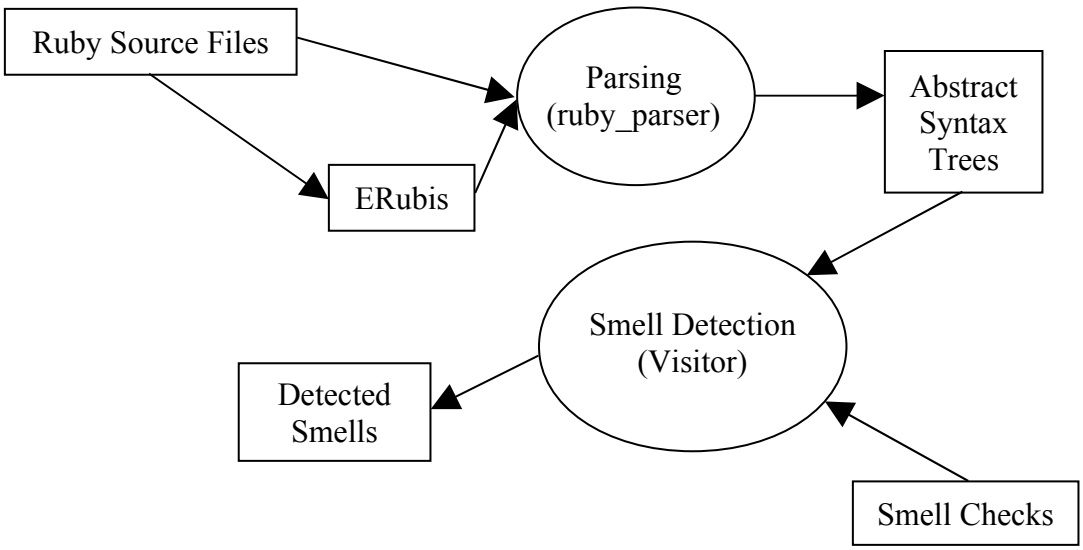


Figure 3.8: High Level Design

3.3 Detector Implementation

This section describes the various components of the smell detection systems and how they operate.

3.3.1 Control Flow Overview

When executing the detector, the Rails application to be analysed should be passed as a command line parameter. The detector first checks that the directory exists and that each of the main Model, Controller and View directories exist.

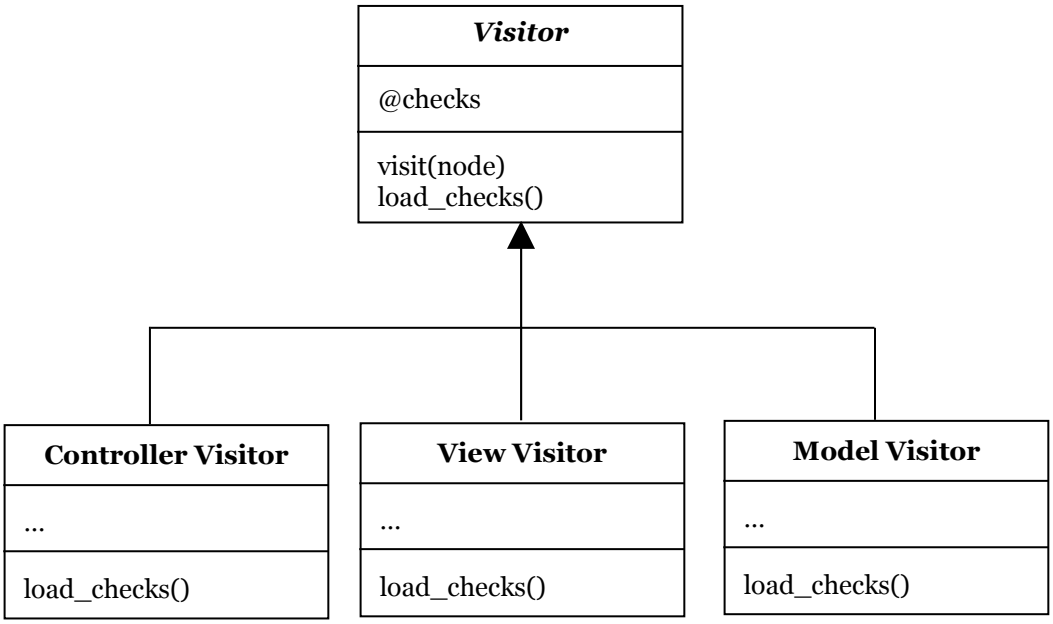
Once these directories are present the detector can proceed. Each of the Model, View and Controller directories are analysed in turn. Every file in each of the directories is loaded into memory with sub-directories checked recursively. A file is read into memory and the appropriate Visitor is created based on the file type. Once the Visitor has been created, an AST is generated from the file contents and is passed to the Visitor to begin the tree traversal.

3.3.2 Visitor Pattern

The Visitor Pattern is a well known Design Pattern that was described in the Gang of Four's book Design Patterns [4]. It is a way of separating an algorithm from the object structure it operates on and allows new operations to be added to the structure without modifying the structure itself.

In this detection tool it is used to allow each of our smell detection algorithms to Visit each node of the AST, check that node and in some cases its sub-nodes for code smells.

A distinct Visitor class was created for each of the three layers of the MVC architecture. Each concrete Visitor class inherits from a parent Visitor class which provides its structure.



Each concrete Visitor instantiates the Smell Checks that are related to the file type being examined.

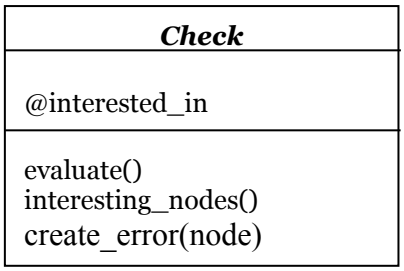
When traversing the AST, the Visitor arrives at each node and allows each Smell Check to examine the node to see if it is interested in checking it for smells by calling its '*interesting_nodes()*' method.

3.3.3 Visitor Factory

A Visitor Factory was created which instantiates the correct Visitor type based on the files type being examined. For example, a Controller Visitor is created for Controllers, a View Visitor for Views and so on.

3.3.4 Smell Checks

The logic for each Smell Check is encapsulated in its own class. A base Check class was created from which each Smell Check algorithm inherits its structure.

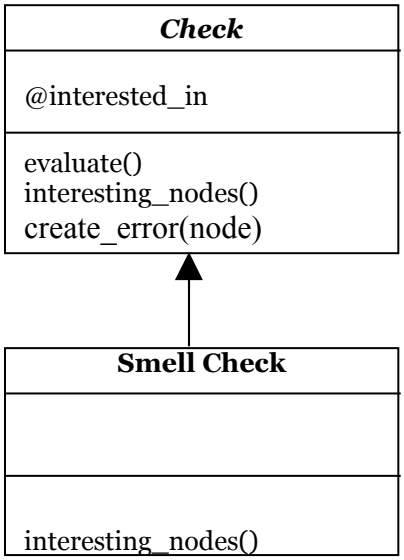


The '*evaluate()*' method is called to execute the actual Smell Check logic. This is done after '*interesting_nodes()*' is called for each Check.

3.3.5 Defining Interesting Nodes

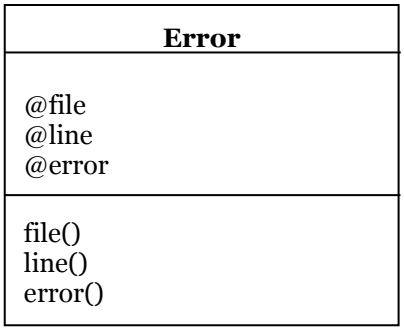
Each Smell Check is only interested in analysing certain node types. It would be very in-efficient for each Smell Check to check every node type it is passed, when it is only interested in certain node types.
Each Smell Check inherits a method called 'interesting_nodes' from its parent class. This method is executed on each Smell Check for every node that is visited. By default the inherited method allows all node types to be analysed by each Smell Check. By overriding the method and specifying the interesting nodes in the '*@interested_in'* array, we can filter certain node types for each Smell Check.

For example, a method line count check would be mainly interested in analysing methods. When the visitor traverses the AST, it checks with each Smell Check to see if it is interested in examining the node that is currently being visited. If it is, the Smell Check is executed. If not the Smell Check is skipped.



3.3.6 Reporting Errors & Smells

Any Smells that are encountered are flagged by creating an instance of an error class. The error class is created by the use of a callback function which is passed to each Smell Checks '*evaluate()*' method. The Error object stores the file name, line number and a smell description.



3.4 Project Structure

This section provides an overview of the directory and file structure of the detection tool that was implemented. The overall application structure and the core files are shown in Figure 3.9.

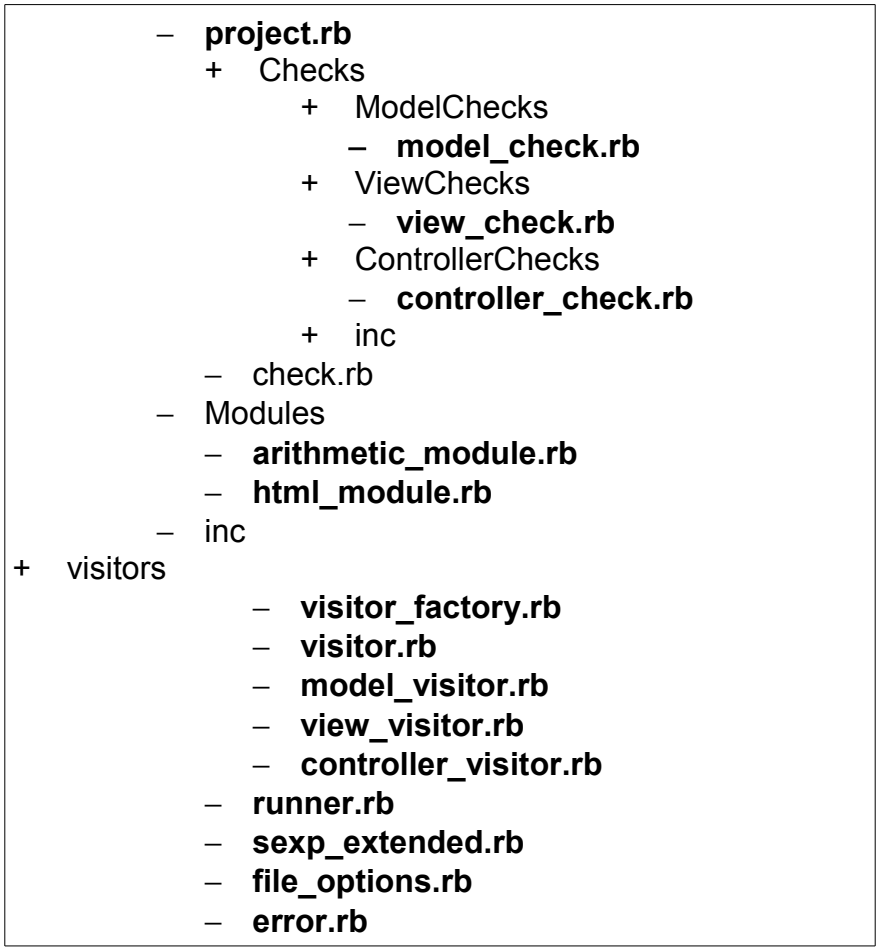


Figure 3.9

3.4.1 Project Structure Breakdown

The project.rb file is located at the root of the tools directory hierarchy. This is the entry point for the tool and is the first file executed when running the detector. Within the root there are three directories.

- Checks
- inc
- Modules

The Checks directory contains each of the implemented smell check algorithms. Within the Checks directory there are a further four directories and a *check.rb* Ruby script. The *check.rb* script is the super class that all implemented checks extend. The three main folders in the Checks directory are ModelChecks, ViewChecks and ControllerChecks which contain the respective Model, View and Controller smell check that were implemented as part of this tool. Within each of these directories is a base class for creating the respective smell checks by

extending that class. So for example, within the ControllerCheck directory is a script called controller_check.rb which extends check.rb. When implementing a smell check for use with Controllers, we extend the ControllerCheck class in controller_check.rb which in turn extends the Check class in check.rb. In this way we can easily identify a specific smell checks class type at runtime if need be. We can also add any common functionality at each of the various layers. Finally, within the Checks directory we have an “inc” folder which is used to hold any modules or classes that may be common to a number of smell checks.

Next in the root directory structure is the “inc” directory. This directory contains some of the core files used by the detection tool. The runner.rb script is responsible for gathering all the files and directories of the target application. It converts each target files source code to an AST representation. It is also responsible for the creation of the appropriate visitor and instructs it to begin traversing the AST. The sexp_extended.rb script is a small class which “monkey-patches” the sexp class and adds in an “accept” method which permits access to each tree node by a visitor class. The file_options.rb script is a container class which holds various file and directory operations that are used by the detection tool. The error.rb script is a simple class which holds details on each of the smells that are encountered while traversing the AST. Finally there is a visitors directory which contains all the visitor classes. There is a VisitorFactory class which creates the appropriate visitor type for the file type being examined. The concrete ModelVisitor, ViewVisitor and ControllerVisitor classes each extend the Visitor class. Each of the concrete visitor classes contains an array of the smell checks that are appropriate to it. When a concrete visitor class is instantiated, it in turn loads up each of the smell checks it has been assigned. So for example, if a new smell check algorithm was implemented we would simply add that smell check to the appropriate visitor class to ensure it is executed.

4. Code Smell Taxonomy

This chapter details some of the more common bad practices that are encountered when developing Ruby on Rails applications. Each of these *Code Smells* were implemented in the Smell Detector as a 'Smell Check'. A number of them are the subject of an evaluation that was later carried out and is outlined in Chapter 5.

This list was compiled based on research of topics such as Skinny Controllers & Fat Models, Rails Best Practices, MVC principles and also in discussion with a number of Rails developers on various forums.

All of the Code Smells presented here can be described as being a 'Primitive Smell Aspect', as defined by Eva van Emden and Leon Moonen in their paper *Java Quality Assurance by Detecting Code Smells*. In this paper they distinguish two types of smell aspects. *Primitive Smell Aspects* are smells that can be observed directly in the code being examined. An example of a primitive smell aspect is "a Controller method contains looping structures". *Derived Smell Aspects* are smells that are inferred from other aspects. An example of such an aspect is "Class C does not use any methods offered by its superclass". [36]

Each Smell Check contains a short description of the smell along with a sample of the node that occurs in the AST. These sample nodes serve as a method of detection for locating the smell.

4.1 Common Controller Smells

The role of a controller is to dictate what to do behind the scenes and what to display in the view. Its primary concern is to delegate tasks. Controllers receive requests, decide what action to take and then delegate that action.
With this in mind, most of the bad practices that are encountered when analysing Controllers are associated with the Controller trying to do too much or containing business logic that should really be located in the Model layer.
The goal is to keep Controllers skinny and ensure they are Intention Revealing.

4.1.1 Looping Structures

The presence of any looping structure, such as for, while, times etc… in a Controller can be an indication of business logic in that Controller and is a candidate for refactoring.

Sample AST Node

```
s(:while, s(:call, s(:gvar, :$i), :<, s(:arglist, ...
```

As detailed in Section 3.1.2, the sample above is a *Call* type node. In it the loop type is identified from the first element in the array. The following loop types and iterators are considered to be bad practice when used in a Controller…

- for
- while
- until
- each
- detect
- times
- upto
- step
- each_index

The smell detection algorithm visits each node in the AST looking for occurrences of looping structures such as those listed. An error is generated if any such instances are located. Care needs to be taken with some iterator types. For example, *respond_to* is classed as an iterator in an AST but is acceptable in Controllers.

4.1.2 Elementary Arithmetic Operations

Elementary arithmetic is the simplified portion of arithmetic and includes the operations of addition, subtraction, multiplication, and division. In addition to the elementary operands we also include other operands such as modulo (%) and power of (^). These types of operations are often used to calculate and return a value and as such are considered as another indication of possible business logic.

Sample AST Node

```
s(:call, s(:call, nil, :y, s(:arglist)), :+, s(:arglist, s(:lit, 10)))
```

The sample above is a *Call* type node. This node represents the basic operation ‘x = y + 10’. In the AST, the arithmetic operand occupies a particular place in the node. We examine each node for these arithmetic operands and generate an error if any are found.

4.1.3 HTML Markup Defined in Controller

One of the main advantages of the MVC architecture is that each layer has distinct responsibilities. Client presentation is the sole responsibility of the View layer and as such any HTML markup should only ever exist in this layer. HTML markup defined in any other layer is a clear contravention of this clean separation of layers that MVC embodies.

Sample AST Node

```
s(:str, "<option></option>")
```

The sample node above is a *Lit* type node. Once located, we take the corresponding value of the node and compare it against a regular expression statement which defines the structure of an HTML tag. If any matches are found an error is generated.

4.1.4 Assigning Variables

Assigning variables is usually done to store state or as an intermediate in calculating a value, both of which are indications that types of business logic may be taking place. The accuracy of this type of smell check is debatable. There are some legitimate cases where a Controller may store a value temporarily as part of its decision making process. However it is also obvious how variable assignment can also be an indicator of business logic taking place.

Sample AST Node

```
s(:lasgn, :sql_condition, s(:str, ""))
```

The sample node above is a *Lit* type node. When identifying the assignment of variables there are a number of nodes that can occur such as *:attrasgn, :attrset, :dasgn_curr, :iasgn, :lasg* and *:masgn.* We also check the type of assignment that is taking place. In the above example it is a string. When we locate an assignment node we check the type of assignment against *:lvar, :ivar, :str* and *:lit* and generate an error if any of these are matched.

4.1.5 Validation Check

In keeping with the OO principle of putting logic where the data it operates on is located, data validation should never be done in a Controller. For example, if the validation to check data before it is inserted into a database table where located in a Controller and another Controller wanted to add data to the same table, it would have to duplicate the data validation which would in turn violate the DRY principle.

The implementation of this check is still quite immature. Currently we only check for instances of regular expression pattern matching taking place. Pattern Matching using regular expressions isn't a firm indicator of validation and when examining the code that triggers this check, we can conclude that not all occurrences are instances of validation. However, pattern matching in a Controller is a possible code smell as it constitutes decision making logic which may be better served by being part of the Model.

Sample AST Node

```
s(:call, s(:ivar, :@question), :match, s(:arglist, ...
```

The sample above uses a *Call* type node. The pattern matching element can be identified by the third element in the array - *:match.*

4.1.6 RESTful Controllers

The RESTful approach is one of the best ways to ensure that Controllers remain skinny. This approach standardises Controllers and provides a structured approach to their construction. Using a RESTful approach is optional and is decided by each individual developer. Not using a RESTful approach does not necessarily mean that a Controller or application is poorly built. However we do make a judgment call that it could perhaps be better built.

Sample AST Node

```
s(:defn, :test_method, s(:args, :var1, :var2), s(:scope, s(:block, s(:nil))))
```

The sample node above is a *Def* type node. As described in Chapter 2, the first element in the array is the node type, *:defn*, which represents a method. The second element in the array is the method name, the third element lists the methods arguments and the fourth element is the method body.

There are three main areas used to detect if a Controller is RESTful. First we check all public method names in the Controller to ensure that only the following are used…

- index
- show
- new
- create
- edit
- update
- destroy

Using the sample node above, the method name contained in position two should be contained in the list above.

Next, we ensure that each method does not take any custom parameters in its declaration. Again, using the sample node above, we count the number of parameters at position three to ensure it is zero.

Finally, we ensure that each method does not return a value and instead closes with the use of the 'respond_to' construct.

4.1.7 Output Check

This Check simply makes sure that a Controller doesn't try to print or output anything. Presentation should be left to the View. There are two main node elements that we are trying to detect here. While visiting the AST, this check is primarily interested in occurrences of statements such as *:print* and *:puts*. Detection of any such statement generates an error.

Sample AST Node

```
s(:call, nil, :puts, s(:arglist, s(:str, "test")))
```

Element three in the *Call* type node above is the element we are interested in with respect to this smell check.

4.1.8 Method Line Count Check

Controllers are composed of methods. A fat Controller has bloated methods. This check simply compares the line count of each method against a set limit and generates an error if the limit is breached. To start with we are looking to identify nodes which represent the beginning of a method.

Arriving at an accurate metric for the maximum allowed number of lines that a method may have is a debatable topic. For the purposes of this check we have decided to set this maximum to twelve lines. Any Controller method with more than twelve lines of code is considered fat.

Currently this check only looks at the total number of lines that it contains. An improvement on this would be to ignore blank lines and to perhaps exclude the 'respond_to' block. Although if this were excluded we would need to drop the maximum allowed number of lines accordingly.

Sample AST Node

```
s(:defn, :test_method, s(:args, :var1, :var2), s(:scope, s(:block, s(:nil))))
```

The number of lines in a method is calculated using the AST's line numbers by simply comparing the start and end line numbers of the method.

4.1.9 Conditionals Check

This check examines each method and counts the number of conditional checks that are taking place. Statements and expressions such as 'if' and 'switch' can be indications of business logic, especially if there is more than one such statement in a single method.

We start by looking for a node that represents the start of a method. Once located, we then traverse the sub-nodes of this node and count the number of matching statements that occur. So we are looking for a combination of nodes in this check. First a *:defn* node and then count the number of *:if* nodes that occur in the sub-nodes.

Sample AST Sub-Node

```
s(:if, s(:call, s(:call, nil, :request, s(:arglist)), ...
```

Basically we are looking for excessive conditional checking here. For each method, a score is kept of how many conditional checks have been discovered. If this score exceeds a set value an error is generated.

4.1.10 Nesting Checks

Besides generally being bad practice, excessive nesting of conditionals and loops can be an indication of business logic. In this check we are looking for nesting of the following conditional statements - *:if, :for* and *:while.* When we encounter an initial occurrence of one of these conditionals we proceed to check its sub-nodes looking for further occurrences.

As we move through the nodes in the AST we keep a score for how deeply nested we are. Nesting to one level has a score of 1, two levels a score of 2 and so on. If the score goes over 2 we take the view that we are too deeply nested.

Sample AST Node

```
(:while, s(:call, s(:call, nil, :i, s(:arglist)), :<, s(:arglist, s(:call, nil, :k, s(:arglist)))), s(:if,
```

In the above sample node we start with a *:while* which triggers the check. Within the sub-nodes is an *:if* node which increases the nesting count and triggers an error.

4.1.11 Use Named Scopes

With Named Scopes the developer is encouraged to use queries defined in a Models methods instead of using the 'find' method directly from a controller. This can encourage the DRY principle and removes sometimes confusing query parameters form the Controller. Rails also has a built in named_scope keyword which allows you to define common query parameters for re-use in multiple queries.

Sample AST Node

```
s(:call, s(:const, :IssueStatus), :find, s(:arglist, s(:lit, :all), ...
```

This Check simply looks for instances of 'find' in Controller methods and suggests that the developer consider using named scopes instead. In the above sample node the key element in the array is element three which contains *:find*.

4.2 Common View Bad Practices

The primary responsibility of the View is to handle how information is presented to the user. Ideally Views should be dumb. A dumb View is passed data and instructed to display it. It should have no idea of where the data has come from or how it was constructed. These smell checks aim to ensure that Views are Dumb in nature.

4.2.1 Variable Assignment

Similar to the Controller check in Section 4.1.4, assigning variables is usually done to store state or as an intermediate in calculating a value, both of which are indications that business logic may be taking place. There are some legitimate cases where a View may store a value temporarily; such as when using an iterator. However it is also obvious how variable assignment can also be an indicator of business logic taking place.

Sample AST Node

```
s(:lasgn, :sql_condition, s(:str, ""))
```

The sample node above is a *Lit* type node. When identifying the assignment of variables there are a number of nodes that can occur such as *:attrasgn, :attrset, :dasgn_curr, :iasgn, :lasg* and *:masgn.* We also check the type of assignment that is taking place against *:lvar, :ivar, :str* and *:lit* and generate an error if any of these are matched.

4.2.2 Excessive Conditional Checks & Loops

In keeping with the principle of Views being Dumb, excessive Conditional Checking and Loops can be an indication that the View is not Dumb enough. Conditional checking should be kept to a minimum in Views to ensure that it contains less Ruby code and more presentation code. Any such checking should be done in the Controller or in a Helper from where it is re-usable across Views and makes them more readable. This check is essentially a combination of the Nesting Check and the Conditionals Checks that is performed on Controllers.

4.2.3 Nesting Check

This is another check that is also performed on Controllers and is discussed in Section 4.1.10 as well as here. Besides generally being bad practice, excessive nesting of conditionals and loops can be an indication of business logic. In this check we are looking for nesting of the following conditional statements - *:if, :for* and *:while.* When we encounter an initial occurrence of one of these conditionals we proceed to check its sub-nodes looking for further occurrences.

As we move through the nodes in the AST we keep a score for how deeply nested we are. Nesting to one level has a score of 1, two levels a score of 2 and so on. If the score goes over 2 we take the view that we are too deeply nested.

Sample AST Node

```
(:while, s(:call, s(:call, nil, :i, s(:arglist)), :<, s(:arglist, s(:call, nil, :k, s(:arglist)))), s(:if,
```

In the above sample node we start with a *:while* which triggers the check. Within the sub-nodes is an *:if* node which increases the nesting count and triggers an error.

4.2.4 Object Creation

Dumb Views should not be creating objects. This should be left to the Controller, the Model or abstracted out to a Helper.

Sample AST Node

```
s(:call, s(:const, :Query), :new, s(:arglist))
```

In the sample above, the *Call* node is used to identify potential nodes that should be examined. The third element in the array (*:new*) is the key element for identifying object creation.

4.2.5 Database Query

Querying a database from the View is also considered bad practice. It is an indication of the View knowing too much about the application logic and also indicates a possible violation of the DRY principle as the query cannot be re-used without it being repeated. Ideally, Views should access databases via the Controller and preferably using a named scope defined on the Model in question.

Sample AST Node

```
s(:call, s(:const, :IssueStatus), :find, s(:arglist, ...
```

The *Call* node type is used here and in particular we need to examine element three in the array which will contain a *:find*.

4.2.6 Elementary Arithmetic Operations

This is another check which is similar to a Controller check described in Section 4.1.2 and is detailed here again. Elementary arithmetic is the simplified portion of arithmetic and includes the operations of addition, subtraction, multiplication, and division. In addition to the elementary operands we also include other operands such as modulo (%) and power of (^). These types of operations are often used to calculate and return a value and as such are considered as another indication of possible business logic.

Sample AST Node

```
s(:call, s(:call, nil, :y, s(:arglist)), :+, s(:arglist, s(:lit, 10)))
```

The sample above is a *Call* type node. This node represents the basic operation ‘x = y + 10’. In the AST, the arithmetic operand occupies a particular place in the node. We examine each node for these arithmetic operands and generate an error if any are found.

4.2.7 Loops Check

Similar to Controllers, looping structures in Views can be, in some cases, bad practice. Obvious caveats to this are looping structures that are used to display the results of a database query.

Sample AST Node

```
s(:for, s(:ivar, :@statuses), s(:lasgn, :new_status), s(:block, s(:call, s(:lvar, :_buf), ...
```

The smell detection algorithm visits each node in the AST looking for occurrences of looping structures such as ‘for’, ‘while’ and ‘times’. AN error is generated if any such instances are located.

5. Evaluation

Building a smell detection system without evaluating its performance is a relatively pointless exercise. This detector was constructed to improve development practices in areas that contained some obvious issues. For this reason we need to perform an evaluation of the detectors results in, as much as possible, a controlled and objective manner.

This chapter presents an empirical study of the effectiveness of the smell detection tool, defined in terms of the accuracy of it's output. An attempt is made to evaluate the tool in terms of its *Precision* [39]. The accuracy of the approach taken here is admittedly open to debate and is subject to threats discussed in Section 5.6.

Section 5.1 examines some common evaluation measures that can be used to measure the effectiveness of this tool while Section 5.2 discusses the approach taken to testing and evaluation. Section 5.3 details the Evaluation phases that comprised this study while Section 5.4 compiles the results of the survey and Section 5.5 summarises these results. Finally Section 5.6 discusses the threats to the results gathered in this evaluation.

5.1 Common Evaluation Measures

The quality of an automated smell detection system can be measured in various ways. In computer science the two most commonly used evaluation metrics are Precision and Recall.

5.1.1 Precision

Precision is defined as the percentage of positive predictions that are correct. It is the measure of the ability of a system to present only relevant items. In the case of the detector being evaluated here, Precision is the percentage of smells a detector locates that are genuine cases of the smell being sought. So, for example, the detector reports that a View contains business logic because it contains elementary arithmetic operations. On inspection of the offending code, is this a valid statement?

5.1.2 Recall

Recall, or Sensitivity, is defined as the percentage of positively labeled instances that were predicted as positive. It is a measure of the number of smells detected that are deemed to be real cases of the smell as a percentage of the number of such smells that are known to exist. So, for example, it would be akin to us

knowing that an application contains 10 elementary arithmetic operation smells in its Views and the detector detecting all 10 of these smells. This would equate to a Recall of 100%.

5.2 Evaluation Approach

This section examines the steps that were taken in order to create an evaluation that was as objective and valid as possible.

5.2.1 Choosing an Evaluation Method

With the evaluation measures decided, there are a number of options available as to how we would proceed with a more objective evaluation of the smell detection tool.

1. Have an independent group of developers analyse an application to identify all the code smells it contains. Then run the detector on it to see how many of these smells it identifies.
2. Use an application that has a known number of smells and see how the detector performs.
3. Run the detector on an application and get independent developers to assess if the smells that were reported are in fact smells.
4. Run the detector on an application and assess ourselves if the smells that were reported are in fact smells.

Each of the options listed have advantages and disadvantages. Only Options 1 and 2 allow us to accurately measure Recall as we would know in advance exactly what smells are present in the application being analysed.
Option 1, while being the preferred option, was eliminated as we did not have access to a group of Ruby volunteers who would be willing to devote significant time to the task and it would be a very onerous task to identify all application smells in advance, especially if looking at a large code base. Option 2 was eliminated as we were unable to find a suitable application that had an agreed list of known smells against which we could evaluate. Option 4 was also ruled out as it would mean we are evaluating the results ourselves and, while this can be a legitimate method, it is not the preferred approach.

This left Option 3 as our best method of evaluating the detector's effectiveness. However we should bear in mind that our ability to assess Recall is severely hampered.

The next step in the evaluation process was the identification of a suitable application and code-base to perform our tests on.

5.2.2 Selecting a Code-Base

Ideally the test application needed to have a number of characteristics.

- It needed to be open-source so we could access the code-base.
- It needs to be relatively large in nature so that we have sufficient code to analyse.
- It must contain a number of smells so that the process is worthwhile.

Through discussions with a number of Rails developers, an open-source application called 'Redmine' [37] was selected. Redmine is a flexible project management web application. The primary reason for its selection is the fact that it was developed relatively early in the Rails lifecycle and as such contains a large number of smells that are perhaps more the consequence of Rails evolving than any fault of the developers. It should be noted here that no judgement is being made on Redmine and it is in fact in the process of being refactored which should eliminate a lot of the smells that were detected.

5.2.3 Defining the Evaluation Criteria

While a personal evaluation of the detectors effectiveness is, in some cases, a perfectly valid approach, we would prefer to get a more objective assessment using experienced Rails developers. A short survey was compiled which contained a number of the more common smells that were detected in the selected code-base.

In the survey each Smell Check was listed along with a short description, a sample of the offending code and a 5 point scale that allowed each developer to indicate whether they agree that this is actually a code smell or not. Each point on the scale was allocated a score which would then be used to calculate the Precision of the Smell Check in question. The 5 point scale and the associated score are shown in Figure 5.1.

Option	Score
Definitely Is	10
Possibly Is	5
No Opinion	0
Possibly Not	-5
Definitely Not	-10

Figure 5.1

An optional comment box was also provided in case any of the participants wanted to elaborate on their answer.

5.2.4 Selecting Code Smells

As each of the smells was to be evaluated by volunteer Rails developers, we wanted to keep the time they would spend on it to a minimum. For this reason,

only a selection of the Smell Checks outlined in Chapter 4 were included in the survey. Seven Smell Checks were selected and are outlined below.

1. Loops in Controllers
2. Elementary Arithmetic Operations in Controllers & Views
3. HTML Markup defined in Controllers
4. Variable Assignment in Views
5. Excessive Conditional Loops & Checks
6. Object Creation in Views
7. Database Query from View

Each of the above Smell Checks was selected based on a number of reasons. For instance, “Excessive Conditional Loops & Checks” was selected as it was one of the more common smells encountered in the selected code-base. While “Object Creation” and “Database Query from View” were selected as the author feels they are strong indications of business logic.

5.3 Evaluation Phases

This section examines the main evaluation phases that were undertaken to assess the performance of the smell detection tool.

5.3.1 Initial Phase

After the core of the smell detection tool had been implemented, we proceeded to add each of the smells that are outlined in Chapter 4 to the detection tool. Note that while all the code smells detailed in Chapter 4 were implemented in the detection tool, only the 7 listed in Section 5.2.4 are the subject of this evaluation.

We tested for each smell as they were being developed. To aid in these initial tests, we took a small Rails application (such as the *Cookbook* application that is packaged with *InstantRails* [38]) and hard-coded into it each of the smells we are looking to detect.

Because of this we were able, to some degree, measure each smell checks recall value during the development. This was possible because we manually added each specific smell type to the trial application. Since the application contained a known specific number of smells, we could evaluate exactly how many of these smells were actually being detected. So from a testing point of view we ensured that the detector had a Recall of 100% for each of the smells before proceeding.

5.3.2 Main Phase

In this section we examine the main evaluation that was carried out on the selected code-base using the criteria defined above. Running the smell detection

tool on Redmine yielded a report containing a large number of smells. The majority of smells outlined in Chapter 4 were detected in varying numbers.

As stated above, the approach taken for evaluation effectively strips us of the ability to measure the recall of each of the smell checks outlined in Chapter 4. While this is true with the majority of the smell checks, there are a number of smell checks that, to some extent, we can comment on with regard to recall. For example, take the RESTful method check. This smell check examines each Controller and determines if it adheres to RESTful principles as outlined in Section 2.5.2. Adhering to this principle can help to keep Controllers skinny. It is however only an optional method of designing Controllers and some developers decide not to adopt the RESTful approach at all. This is a perfectly valid choice to make. However we can state quite clearly if a Controller is RESTful or not and as such can accurately calculate recall for this check. In the case of Redmine, none of its Controllers were found to be RESTful in nature and therefore we can state that the recall of the RESTful smell check is 100%. However since adopting RESTful principles is optional, this smell check was intentionally omitted from this evaluation.

The next step in the evaluation process was an assessment of the detected smells to determine whether or not they actually represented the smell in question (i.e. their precision). This was achieved by distributing the survey to a number of experienced Rails developers and compiling their results into meaningful figures.

5.4 Compiling the Survey Responses

The survey was distributed to 10 Rails developers including a number of developers of existing Ruby smell detectors. The following sections are the compiled results of the survey along with a summary of the comments received.

Only Precision is calculated for each of the smell checks presented here. Recall was not calculated as we were unable to compile complete numbers on the numbers of each smell present in the application as a whole. We score each smell check on its precision out of 100% using the 5 point scale detailed above.

For an explanation of each of the smell checks listed please refer to Chapter 4. Appendix A contains the actual sample code that was extracted from Redmine for each of the smells evaluated.

5.4.1 Loops in Controllers

The responses gathered contained 8 “Definitely Is” replies and 2 “Possibly Is” replies. Using the scale in Figure 5.1 this equates to a Precision of 90%.

The comments received ranged from declarations of agreement that this does represent a code smell to suggestions that the code should be placed in the Model or separate modules.

5.4.2 Elementary Arithmetic Operations in Controllers & Views

The responses gathered contained 4 “Definitely Is” replies, 5 “Possibly Is” replies and 1 “Possibly Not” reply. Using the scale in Figure 5.1 this equates to a Precision of 60%.

The comments received ranged from declarations of agreement that this does represent a code smell to suggestions that the code should be placed in the Model, separate modules or Helpers. There was some agreement that this smell check could be dependant on the context of the code and that there are some cases were this was permissable in Views.

5.4.3 HTML Markup in Controllers

The responses gathered contained 10 “Definitely Is” replies. Using the scale in Figure 5.1 this equates to a Precision of 100%.

The comments received were all in complete agreement that this is a valid code smell.

5.4.4 Variable Assignment in Views

The responses gathered contained 5 “Definitely Is” replies and 4 “Possibly Is” replies and 1 “Possibly Not” reply. Using the scale in Figure 5.1 this equates to a Precision of 65%.

The comments received ranged from declarations of agreement that this does represent a code smell to suggestions that the code should be placed in a Helper. The general feeling was that this is a common issue among developers who don’t like to create variable in Views but can’t see a clean and clear solution.

5.4.5 Excessive Conditional Loops & Checks

The responses gathered contained 9 “Definitely Is” replies and 1 “Possibly Not” reply. Using the scale in Figure 5.1 this equates to a Precision of 85%.

The comments received ranged from declarations of agreement that this does represent a code smell to suggestions that the code should be placed in a Helper. There was a general feeling that sometimes doing this is unfortunately unavoidable, but all agreed that it was not an optimal solution. This perhaps points to areas in Rails that could be improved.

5.4.6 Object Creation in Views

The responses gathered contained 3 “Definitely Is” replies and 7 “Possibly Is” replies. Using the scale in Figure 5.1 this equates to a Precision of 65%.

The comments received ranged from declarations of agreement that this does represent a code smell to suggestions that perhaps it is valid depending on the context of the code in question.

5.4.7 Database Query from Views

The responses gathered contained 6 “Definitely Is” replies and 4 “Possibly Is” replies. Using the scale in Figure 5.1 this equates to a Precision of 80%.

The comments received ranged from declarations of agreement that this does represent a code smell to suggestions that perhaps this is fine in certain contexts.

5.5 Summary

This section summarises the results gathered in Section 5.4. To recap here are the 7 code smells that were selected for this evaluation.

1. Loops in Controllers
2. Elementary Arithmetic Operations in Controllers & Views
3. HTML Markup defined in Controllers
4. Variable Assignment in Views
5. Excessive Conditional Loops & Checks
6. Object Creation in Views
7. Database Query from View

Figure 5.2 summarises the results of the survey with regard to the precision of each smell check.

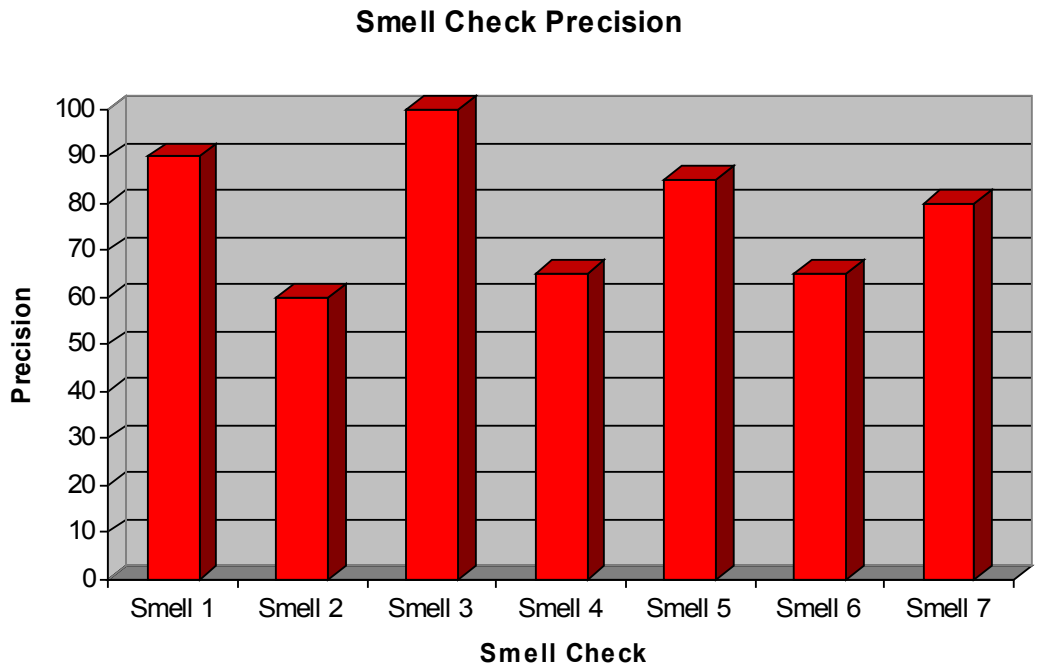


Figure 5.2

5.6 Threats to Validity

The data presented in the previous sections aims to prove the usefulness of the smell detection tool in detecting the specified Code Smells and in turn accurately detecting the fat controller anti-pattern. However the validity of this approach can

legitimately be called into question. For instance, we might conclude that the results of this study (which was done on a specific application, with certain types of people) can be generalised to another context (for instance, another application, with slightly different people). Is this a valid conclusion?

In scientific research, validity refers to whether a study is able to scientifically answer the questions it is intended to answer. There are various forms of validity that can be analysed. Construct Validity, for example, is an analysis of whether or not the tests developed from a theory, do actually prove what the theory claims they should. For instance, do IQ tests actually measure intelligence?

A full discussion on validity is, however, beyond the scope of this work. In this section we are mainly concerned with what the threats are to the results presented above. With this in mind we discuss some of the more obvious threats below.

The first threat that needs to be highlighted is that of the volunteer Rails developers. This study was conducted using 10 participants. This is a relatively small sample to draw definitive conclusions from. Also, while each of the participants is an experienced Rails developer, we cannot claim that they are each experts in the field and as such their input cannot be considered infallible.

Secondly, the Code Smells highlighted above have only been tested on a single Rails application. We can therefore only make assumptions about the benefits of using the tool on other Rails applications. Also, each Code Smell is looked at, to some extent, in isolation from each other and without the overall goal in mind. For example, while the study may have yielded a generally positive response in terms of the Precision of each Code Smell, no mention is made of how these feed into the identification of the overall Anti-Pattern. Another issue is that this evaluation only assesses a selection of the overall catalogue of Code Smells discussed in Chapter 4. We can therefore not draw any sort of conclusions on the effectiveness of those Code Smells.

The possibility exists that each of these Code Smells could be detected without the target Anti-Pattern being present. For instance, does the identification of these Code Smells guarantee the presence of the Anti-Pattern?
These are just some of the issues that pose a threat to the validity of this study. The author accepts and welcomes that many more threats could also be raised.

6. Conclusion and Further Work

This chapter will conclude this work by restating its aims, summarizing its achievements and suggesting future work that could be of interest in extending the concepts discussed in this report.

6.1 Conclusion

6.1.1 Theme and Motivation

This work stemmed from the practice of Refactoring and was primarily motivated by Anti-Patterns that have become evident in the development of applications using Web Application Frameworks such as Ruby on Rails. The main focus of this work was the identification of what we termed the Fat Controller Anti-Pattern. We suggested that this anti-pattern can be diagnosed by the presence of a number of code smells.

Chapter 2 examined the background and discussed the principle of Skinny Controllers, Fat Models and Dumb Views. We proceeded to suggest that not adhering to this principle is a root cause of the anti-pattern and as such is considered to be bad practice and a candidate for refactoring. Chapter 2 then went on to offer possible methods of detecting the anti-pattern via the concept of a code smell and examined how static analysis of code smells can aid in this process. Chapter 4 detailed the various smells that were identified as being contributors to aid in the identification of the Fat Controller anti-pattern and we discussed possible methods of detection.

6.1.2 Automated Smell Detection

An automated static analysis tool was developed which examines a code-base for instances of the smells outlined in Chapter 4. Instances of these smells are then used to infer that the Fat Controller Anti-Pattern could be present in the application being analysed.

We evaluated the detection tool by running trials on an open-source Rails application called Redmine. The results of these trials were presented in Chapter 5. In that chapter we detail a survey that was completed by a number of experienced Rails developers. In terms of Smell Check Precision, the results garnered were encouraging.

There is clearly a limit to how accurately we can assess the quality of an application with regard to the Fat Controller Anti-Pattern. However, the results presented in Chapter 5 suggest some merit in ensuring good practice, when

developing web applications, if used in conjunction with the smell detection tool and its component smell checks.

6.2 Future Work

In this section we will present some ideas on how the smell detection system might be improved and some more general ideas of related areas that may be worth investigating.

6.2.1 Enhancing and Extending

The instances of Code Smells outlined in this work, is by no means complete. Each of the Smell Checks discussed is primarily aimed at detecting the improper placement of business logic in an application. It is likely that there are many more possible Code Smells that could be used to identify business logic and in turn aid in the detection of the Fat Controller anti-pattern.

Equally, the Smell Checks that are discussed and implemented here could be refined to potentially increase their Precision and Recall. For example, with regard to the RESTful Controller check, it could be enhanced by examining custom routes in the *routes.rb* file.

Another enhancement that could be considered would be to package this smell detection tool up as RubyGems package which could be installed using the “gem install” functionality.

6.2.2 IDE Integration

The smell detection tool implemented here is accessed via a Ruby command prompt. While not ideal, it is satisfactory within the scope of this work. An obvious improvement on this would be its integration into an IDE such as Eclispe, where its alerts could be delivered in real-time and in a fashion that is familiar to users of the IDE.

6.2.3 Other Applications of these Concepts

While the concepts discussed in this work focus on the Ruby on Rails framework, they are by no means limited to this framework. Each of the Code Smells outlined attempts to identify misplaced business logic in the context of where it should ideally be located when developing with a framework that utilises the MVC architecture. Ruby on Rails is not the only Web Application Framework which utilises MVC and as such the concepts outlined here could possibly be applied to other frameworks utilising the MVC framework.

6.3 In Conclusion

Web Application Frameworks enable the developer to work more efficiently and with greater structure by removing the burden of “plumbing” that previously plagued web application development. However, while a framework defines the scaffolding upon which an application is built, it is up to the developer to flesh out this scaffolding by adding the target applications business logic. It is here that problems can arise, if developers favour rapid implementation of application logic over correct structure, resulting in the bad development practice that the Fat Controller Anti-Pattern embodies.

Using the concepts and tools outlined in this work, problems can be indicated to developers in an interactive and potentially real-time fashion. Refactoring tasks that may previously have been tedious and error-prone can become more accessible and efficient, ultimately resulting in improved application development.

Appendix A

The following are the code segments that were extracted from the Rails application Redmine in support of the Evaluation that was carried out in Chapter 5.

5.4.1 Loops in Controllers

```
while date_from <= @to.to_time && @periods.length < 100
  case @columns
  when 'year'
    @periods << "#{date_from.year}"
    date_from = (date_from + 1.year).at_beginning_of_year
  when 'month'
    @periods << "#{date_from.year}-#{date_from.month}"
    date_from = (date_from + 1.month).at_beginning_of_month
  when 'week'
    @periods << "#{date_from.year}-#{date_from.to_date.cweek}"
    date_from = (date_from + 7.day).at_beginning_of_week
  when 'day'
    @periods << "#{date_from.to_date}"
    date_from = date_from + 1.day
  end
end
```

5.4.2 Elementary Arithmetic Operations in Controllers & Views

```
Controller…

@total_hours = @hours.inject(0) {|s,k| s = s + k['hours'].to_f}

View…

@gantt.zoom.times { zoom = zoom * 2 }
(<%= '%0.0f' % (version.closed_issues_count.to_f /
version.fixed_issues.count * 100) %>%)
```

5.4.3 HTML Markup in Controllers

```
content_tag('select', '<option></option>' +
  version_options_for_select(
    @project.shared_versions.open, @version),
  :id =>  'issue_fixed_version_id',
  :name => 'issue[fixed_version_id]')
```

5.4.4 Variable Assignment in Views

```
zoom = 1
@gantt.zoom.times { zoom = zoom * 2 }

subject_width = 330
header_heigth = 18

headers_height = header_heigth
show_weeks = false
show_days = false

if @gantt.zoom >1
    show_weeks = true
    headers_height = 2*header_heigth
    if @gantt.zoom > 2
        show_days = true
        headers_height = 3*header_heigth
    end
end
```

5.4.5 Object Creation in Views

```
<% colors = Hash.new {|k,v| k[v] = (k.size % 12) } %>
```

5.4.6 Database Query from Views

```
<% projects = Project.active.find(:all, :order => 'lft') %>
```

5.4.7 Excessive Conditional Loops & Checks

```
Controller…

# quick jump to an issue
if @question.match(/^#?(\d+)$/) && Issue.visible.find_by_id($1.to_i)
    redirect_to :controller => "issues", :action => "show", :id => $1
    return
end

@object_types = Redmine::Search.available_search_types.dup
if projects_to_search.is_a? Project
    # don't search projects
   @object_types.delete('projects')
    # only show what the user is allowed to view
   @object_types = @object_types.select {|o|
        User.current.allowed_to?("view_#{o}".to_sym,
projects_to_search)}
end

View…

<% if show_weeks
  left = 0
  height = (show_days ? header_heigth-1 :
    header_heigth-1 + g_height)
  if @gantt.date_from.cwday == 1
    # @date_from is monday
    week_f = @gantt.date_from
  else
  # find next monday after @date_from
  week_f = @gantt.date_from + (7 - @gantt.date_from.cwday + 1)
  width = (7 - @gantt.date_from.cwday + 1) * zoom-1
  %>
  <div style="left:<%= left %>px;top:19px;width:
  <%= width %>px;height:<%= height %>px;"
    class="gantt_hdr"> </div>
  <%
  left = left + width+1
  end %>
  <%
  while week_f <= @gantt.date_to
    width = (week_f + 6 <= @gantt.date_to) ?
      7 * zoom -1 : (@gantt.date_to - week_f + 1) * zoom-1
  %>
  <div style="left:<%= left %>px;top:19px;width:
  <%= width %>px;height:<%= height %>px;" class="gantt_hdr">
  <small><%= week_f.cweek if width >= 16 %></small>
  </div>
  <%
  left = left + width+1
  week_f = week_f+7
  end
end %>
```

Appendix B

This section contains details on how to run the code smell detection tool that was implemented as part of this work. These instructions assume that the tool is being run on a Microsoft Windows operating system.

1. The 'project.rb' file is the entry point for execution of this tool.
2. The detection tool is executed from the command line. This requires either Ruby to be specified in the Path or to execute the Ruby command from within the correct directory.
3. The directory of the Rails application that is to be examined should be passed inas a parameter when executing the tool.
4. There are a number of dependencies for this toll to work correctly. These dependencies take the form of gems that need to be installed first. The following gems should be installed using the 'gem install' command.

    - ruby_parser
    - erubis

References

[1] Ruby on Rails.
http://rubyonrails.org

[2] Opdyke, William. (1992). Refactoring Object-Oriented Frameworks. Ph.D. Thesis. University of Illinois at Urbana-Champaign.

[3] Fowler, Martin. (1999). Refactoring: Improving the Design of Existing Code. Addison Wesley.

[4] Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). Design Patterns: Elements of Reusable Object Oriented Software. Addison Wesley.

[5] Model-View-Controller.
http://msdn.microsoft.com/en-us/library/ff649643.aspx

[6] Web application framework
http://en.wikipedia.org/wiki/Web_application_framework

[7] Java EE. Java Enterprise Edition.
http://www.oracle.com/us/javaee/index.html

[8] CakePHP.
http://cakephp.org

[9] Django.
http://www.djangoproject.com

[10] Spring.
http://www.springsource.org/

[11] Model View Presenter.
http://msdn.microsoft.com/en-us/magazine/cc188690.aspx

[12] N-Tier Data Applications Overview
http://msdn.microsoft.com/en-us/library/bb384398.aspx

[13] Trygve, M.H. (1979) Reenskaug/MVC, Xerox PARC.
http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

[14] Burbeck, S. (1987, 1992). How to use Model-View-Controller.
http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html

[15] Dijkstra, Edsger. (1982). On the role of Scientific Thought. http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html

[16] Designing Enterprise Applications with the J2EE Platform. http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/app-arch/app-arch2.html

[17] Loud Thinking http://www.loudthinking.com/about.html

[18] The Philosophy of Ruby http://www.artima.com/intv/ruby.html

[19] Convention over Configuration http://www.sonatype.com/books/mvnref-book/reference/installation-sect-conventionConfiguration.html

[20] Hunt, A., Thomas, D. (1999). The Pragmatic Programmer: From Journeyman to Master. Addison Wesley.

[21] Web Services Glossary. http://www.w3.org/TR/ws-gloss/

[22] Koenig, A. (1995). Patterns and Antipatterns. Journal of Object-Oriented Programming. Cambridge University.

[23] Brown, W., Malveau, R., McCormick, S.,Mowbray, T., Thomas, S. (1998). AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons.

[24] RailsConf Recap: Skinny Controllers http://www.therailsway.com/2007/6/1/railsconf-recap-skinny-controllers

[25] Buck, J. (2006). Skinny Controller, Fat Model http://www.therailsway.com/2007/6/1/railsconf-recap-skinny-controllers

[26] Model View Controller http://c2.com/cgi/wiki?ModelViewController

[27] ESC/Java2. Extended Static Checker for Java Version 2. Kind Software. http://sort.ucd.ie/products/opensource/ESCJava2/

[28] Meyer, B. (1994). Object Oriented Software Construction. Prentice Hall.

[29] parse_tree & ruby_parser. seattle.rb. http://parsetree.rubyforge.org/

[30] Rutherford, K. Reek. https://github.com/kevinrutherford/reek/wiki/

[31] Andrews, M. Roodi. https://github.com/martinjandrews/roodi

[32] Flay. https://github.com/seattlerb/flay

[32] Flog. https://github.com/seattlerb/flog

[33] Erubis. http://www.kuwata-lab.com/erubis/

[34] Merb. http://www.merbivore.com/index.html

[35] Room 101. Monkey Patching.
http://gbracha.blogspot.com/2008/03/monkey-patching.html

[36] Emden, Eva van., Moonen, Leon. Java Quality Assurance by Detecting Code Smells.

[37] Redmine
http://www.redmine.org/

[38] Instant Rails
http://instantrails.rubyforge.org/wiki/wiki.pl

[39] Basic Evaluation Measures for Classifier Performance
http://webdocs.cs.ualberta.ca/~eisner/measures.html

[40] Martin, James. (1983). Managing the Database Environment.
Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1983.

Bibliography

Cooper, I. (2008). The Fat Controller
http://codebetter.com/iancooper/2008/12/03/the-fat-controller/

RailsConf Recap. (2007). Skinny Controllers
http://www.therailsway.com/2007/6/1/railsconf-recap-skinny-controllers

Marai, C. (2010). Rails Best Practices: Fat Model – Skinny Controller
http://blog.devinterface.com/2010/06/rails-best-practices-1-fat-model-skinny-controller/

Model-View-Controller
http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

Holm, M. (2010). Practicing Ruby
http://blog.rubybestpractices.com/posts/judofyr/sexp-for-rubyists.html

(2002). Model-View-Controller Pattern
http://www.enode.com/x/markup/tutorial/mvc.html

Model View Controller
http://c2.com/cgi/wiki?ModelViewController

What is a Software Framework? And why should you like ‘em?
http://info.cimetrix.com/blog/bid/22339/What-is-a-Software-Framework-And-why-should-you-like-em

History of Web Frameworks
http://www.flickr.com/photos/mraible/4378559350/

What is a Software Framework?
http://www.maxxess-systems.com/whitepapers/what_is_a_software_framework.pdf

Why use an Architectural Framework?
http://www.jcorporate.com/expresso/doc/edg/edg_WhyUseFramework.html

Web Application Frameworks
http://en.wikipedia.org/wiki/Web_application_framework

J2EE Architecture Approaches
http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/app-arch/app-arch2.html

Code Smell
http://en.wikipedia.org/wiki/Code_smell

CRUD
http://en.wikipedia.org/wiki/Create,_read,_update_and_delete

Java Quality Assurance by Detecting Code Smells
http://www.google.ie/url?sa=t&source=web&cd=1&ved=0CBQQFjAA&url=http%3A%2F%2Fciteseerx.ist.psu.edu%2Fviewdoc%2Fdownload%3Fdoi%3D10.1.1.80.2066%26rep%3Drep1%26type%3Dpdf&rct=j&q=code%20smell%20aspects&ei=jc-ATZ6LFoOAhQfFy72aBw&usg=AFQjCNH12K99GdE18Z7aY4n2kSJHap18lA

Convection over Configuration (CoC)
http://en.wikipedia.org/wiki/Convention_over_Configuration

Don’t Repeat Yourself (DRY)
http://en.wikipedia.org/wiki/Don%27t_Repeat_Yourself

Program Analysis and Understanding
http://www.cs.umd.edu/class/fall2010/cmsc631/data-flow.pdf

Static Analysis for Ruby
http://www.codecommit.com/blog/ruby/pipe-dream-static-analysis-for-ruby

Static Analysis Tools Roundup: Roodi, Rufus, Reek, Flay
http://www.infoq.com/news/2008/11/static-analysis-tool-roundup

Philosophy and Practical Implementation of Static Analyzer Tools
http://www.dse.nl/~thelosen/artikelen/static_analysis.pdf

REST vs SOAP Web Services
http://www.petefreitag.com/item/431.cfm