University of Minho
Informatics Department

Master Course in Informatics Engineering

The Role of Best Practices in Assessing Software Quality

Miguel Regedor
University of Minho

Supervised by:

Main Supervisor Pedro Rangel Henriques, from University of Minho.

Co-Supervisor Daniela da Cruz, from University of Minho.

Braga, 27th June 2013

Abstract

This document presents a master thesis in Computer Science, in the area of Program Comprehension and Static Code Analysis. This work (thesis preparation and writing) is a component of the second year of the Masters degree, that will be achieved in University of Minho at Braga, Portugal.

Thousands of open source software (OSS) projects are available for collaboration in platforms like Github or Sourceforge. However, like traditional software, OSS projects have different quality levels. The developer, or the end-user, needs to know the quality of a given project before starting the collaboration or its usage—they may trust in the package before making a decision.

In the context of OSS, trustability is a much more sensible concern; mainly end-users usually prefer to pay for proprietary software in order to feel more confident in the package quality. OSS projects can be assessed like traditional software packages using the well known software metrics.

In this document we want to go further and propose a finer grain process to do such quality analysis, precisely tuned for this unique development environment. As it is known, along the last years, open source communities have created their own standards and best practices*. Nevertheless, the classic software metrics do not take into account the* best practices *established by the community. The notion that it could be worthwhile to consider this peculiarity as a complementary source of assessment data was the essence of this thesis.*

Taking Rails OSS community and its projects as framework, this document discusses the role of best practices *in measuring software quality and describes the studies carried out, to build a new code analysis tool that will enable users to make better choices about what software to use and help developers to improve their software.*

Resumo

Este documento apresenta uma tese de mestrado em Ciência da Computação, na área de compreensão do programas e análise estática de código.

Este trabalho integra o plano curricular do segundo ano de mestrado, e será defendido na Universidade do Minho em Braga, Portugal.

São milhares os projetos de software open source disponíveis para colaboração em plataformas como o Github ou Sourceforge. No entanto, tal como o software tradicional, os projetos OOS diferem entre si a nivel de qualidade. Neste sentido, surge a necessidade do programador ou o utilizador final, saber o grau qualidade de um determinado projecto, antes mesmo de decidir iniciar uma colaboração ou o seu uso. Sendo que para este efeito deve existir confiança no pacote de software, antes de qualquer tipo de decisão sobre a matéria.

No contexto de software open source a confiabilidade surge como uma preocupação relevante, sendo que, muitas das vezes os utilizadores finais preferem adquirir software proprietário com o intuito de sentirem níveis de confiança mais elevados em relação ao referido pacote de software. Mais se refere que a avaliação de projetos de software open source pode ser executada de forma idêntica à de software tradicional, utilizando métricas de software de conhecimento geral.

Neste documento, pretendemos ultrapassar este ponto e propor um novo processo para fazer a dita análise de qualidade, ajustado precisamente para este ambiente de desenvolvimento de software único. Como é sabido, ao longo dos últimos anos, as comunidades de código aberto têm criado suas próprias regras e boas práticas*. No entanto, as métricas de software clássicas não têm em conta as* boas práticas *estabelecidas pela comunidade. Nós sentimos que poderia valer a pena considerar essa peculiaridade como uma fonte complementar de dados de avaliação.*

Tomando como base de trabalho a comunidade open source do framework Ruby on Rails, o presente artigo discute o papel das boas práticas *na medição da qualidade de software, descreve os estudos realizados e a construção de um novo código de ferramenta de análises que permitirá aos usuários optar por melhores soluções, nomeadamente no que se refere ao tipo de software a utilizar e ainda ajudar os developers a melhorar a qualidade do seu software.*

Acknowledgements

Firstly, I want to thank Prof. Dr. Pedro Rangel Henriques and Prof. Dr. Daniela da Cruz for accepting to be my advisors and for always pushing me to work harder. Their support and guidance was of most value to this work.

Secondly, I would like to thank my mother for the constant encouragement throughout my studies.

Thank you to all my friends, Taekwondo colleges and students, for their friendship and endless support. My work colleges at ProSiebenSat1 and also 5 exceptional people that together with me, before I moved to Munich, accepted the challenge of starting a company called Group Buddies.

Although not personally acquainted, I would like to thank Richard Huang, he started the rails best practices project, which I used as base for developing my work.

Last but not the least, thanks to the Rails community and everyone that read this thesis and contributed with corrections and critics.

Contents

Contents iii

List of Figures vii

List of Tables ix

List of Listings xi

1 Introduction 3

2 Open Source 7
- 2.1 The Open Source Movement . . . 7
- 2.2 Open Source Software Development . . . 10
- 2.3 Open Source Project Hosting Platforms . . . 11
- 2.4 The Need to Assess Open Source Projects . . . 12

3 Ruby on Rails 15
- 3.1 Ruby Programing Language . . . 15
- 3.2 Ruby on Rails Framework . . . 18
- 3.3 Model-View-Controller . . . 19
- 3.4 Convention over Configuration . . . 20
- 3.5 Don't Repeat Yourself . . . 21
- 3.6 The Framework Structure . . . 21
  - 3.6.1 Active Record . . . 22
  - 3.6.2 Action Pack . . . 22
  - 3.6.3 Action Mailer . . . 23
  - 3.6.4 Active Support . . . 23
- 3.7 Rails Community . . . 23

4 Software Metrics 25
- 4.1 Assessing Open Source Software . . . 25
- 4.2 Classic Software Metrics . . . 27
  - 4.2.1 Lines of Code . . . 27
  - 4.2.2 Cyclomatic Complexity . . . 28

4.2.3 Fan-In and Fan-Out . . . 28
4.2.4 Object-Oriented Metrics . . . 28
4.3 Available Tools . . . 29

5 Best Practices **31**
5.1 Best Practices in Open Source Software Projects . . . 31
5.2 Identifying Best Practices . . . 32
5.3 Best Practices Examples . . . 34
5.4 Ruby on Rails Best Practices . . . 35

6 Assessing Ruby on Rails Projects **39**
6.1 First Study . . . 39
6.2 Second Study . . . 41
6.3 Results . . . 42

7 Conclusion **47**

References **51**

A Rails Best Practices **53**
A.1 Remove Tab . . . 53
A.2 Remove Trailing Whitespace . . . 53
A.3 Add Model Virtual Attribute . . . 54
A.4 Always Add Db Index . . . 54
A.5 Dry Bundler In Capistrano . . . 54
A.6 Isolate Seed Data . . . 55
A.7 Keep Finders On Their Own Model . . . 55
A.8 Law Of Demeter . . . 55
A.9 Move Code Into Controller . . . 56
A.10 Move Code Into Helper . . . 56
A.11 Move Code Into Model . . . 56
A.12 Move Finder To Named Scope . . . 57
A.13 Move Model Logic Into Model . . . 57
A.14 Needless Deep Nesting . . . 57
A.15 Not Use Default Route . . . 58
A.16 Not Use Time Ago In Words . . . 58
A.17 Overuse Route Customizations . . . 58
A.18 Protect Mass Assignment . . . 59
A.19 Remove Empty Helpers Review . . . 59
A.20 Remove Unused Methods In Controllers . . . 59

A.21 Remove Unused Methods In Helpers 59
A.22 Remove Unused Methods In Models 60
A.23 Replace Complex Creation With Factory Method 60
A.24 Replace Instance Variable With Local Variable 60
A.25 Restrict Auto Generated Routes 60
A.26 Simplify Render In Controllers 61
A.27 Simplify Render In Views 61
A.28 Use Before Filter 61
A.29 Use Model Association 62
A.30 Use Multipart Alternative As Content Type Of Email 62
A.31 Use Observer 62
A.32 Use Query Attribute 63
A.33 Use say and say_with_time in migrations to make a useful migration log 63
A.34 Use Scope Access 63

List of Figures

3.1 Model-View-Controller Diagram 19
3.2 Rails Project File Structure 22

5.1 Rails Best Practices (best practice discussion page) 36

6.1 Ruby on Rails Best Practices Report 45

List of Tables

2.1 Open source project hosting websites 12

6.1 Best practices analyzer gem raw results using 7 open source projects 40
6.2 Best practices analyzer gem normalized results using 7 open source projects . . . 41
6.3 Best practices analyzer gem results using 40 open source projects 42
6.4 Relations between NPBs forks and watchers 43

List of Listings

3.1 Sample Ruby Code 16
3.2 Sample Ruby Code (Rails Environment) 17
3.3 URL Example 20
3.4 Sample Model file “product.rb” 21
3.5 Command to create a new rails application 21
3.6 Cumber feature 23

This thesis is dedicated to my mother.

Chapter 1

Introduction

Nowadays, Open Source Software (OSS) is well disseminated. Thousands of OSS packages can be found online and are free to download, in Open Source Project Hosting Websites (OSPHW) like SourceForge[1], Google Code[2], or GitHub [3]. Those websites, usually in conjunction with a Version Control System (VCS), make it easy for developers all around the globe to collaborate in Open Source Software Projects (OSSP). They also act as a way to make software available to users.

The OSS establishment is clear; according to NetCraft[4], the market share for top servers across the million busiest sites was 66.82% for the open source web server Apache, much higher than the 16.87% for Microsoft web servers in May 2010. Even governments started noticing open source during the last few years, and in some cases adopted it[Hah02]. The broad acceptance of OSS means that now OSS is used by more people than just computer specialists.

John Powell[5] has declared that by measuring the savings that people are making in license fees, the open-source industry is worth 60 billion dollars. Matt Asay[6] shares that from the customer's perspective, open source can be now considered the largest software industry in the world. The full review can be found at CNET News[7].

Usually large industries have a strict organization model, that is not the way open source communities operates. Open Source communities work in a kind of *bazaar style*. [RE00] compares the traditional software development process to build cathedrals, consisting of a few specialized individuals working in isolation. Open source development seemed to resemble a great babbling *bazaar*. But OSS is not developed all the time in *bazaar style*, and each community can have particular habits. Currently, big open source projects can have companies supporting them. However, most projects are not that big, and sometimes it is hard to distinguish the project developers from the project customers/users. Because of this, bug reports and wanted features can become indistinguishable, too. The specification of an open source software project evolves in an organic

[1] http://sourceforge.net/.
[2] http://code.google.com/.
[3] https://github.com/.
[4] http://news.netcraft.com/archives/2010/05/14/may_2010_web_server_survey.html/, accessed on 2010/12/21.
[5] John Powell is CEO, President, and Co-founder, Alfresco Software Inc.
[6] Matt Asay is chief operating officer at Canonical, the company behind the Ubuntu Linux operating system.
[7] http://news.cnet.com/8301-13505_3-9944923-16.html/ accessed on 2010/12/21.

way [CM07].

Can software that is developed in such chaotic way be trusted as a high quality product? Shockingly, *bazaar style* seemed to work [HS02]. Some big projects, for instance Linux distributions such as Ubuntu[8], are proof. However, how can the quality of this software be measured?

The most basic concept of software quality is commonly recognized as lack of "bugs", and the meeting of the functional requirements. But quality is not simply based on that [GKS+07]. The quality of a software system depends, among other things, on update frequency, quantity of documentation, test coverage, number and type of its dependencies and good programming practices. By analysing those parameters a user can make a better choice when selecting software for a specific task [MA07].

When a user/developer finds a new OSSP, for example in GitHub, the things that will most influence the time needed to have a better understanding of the project, to use, or to collaborate in it are the quality of the documentation and the source code readability. Although the OSPHWs provide plenty of useful information about the hosted projects, they do not currently give a quick answer to the following questions: Does this project have good documentation? Does the code follow standards? How similar is it to other projects?

An OSSP is built up from hundreds, sometimes thousands, of files. It can be coded in many different computer languages. To manually analyze a software project is a very hard and time-consuming task, and not all users have the ability to answer the previous questions by looking at the source code [CAH03].

With that in mind, a system capable of analyzing and measuring a given OSSP, producing detailed quantitative and qualitative reports about it, would enable users to make better choices and developers to further improve the package. There is no doubt regarding the benefits of such system but to create this system, many things should be taken into consideration that can become the subject of huge discussion.

Open source communities are constantly creating and improving their working methodologies. And even without noticing, communities create rules and best practices. This document supports the idea that by following *best practices*, software projects increase their maintainability level, which is a quality attribute of great importance for OSSP. This assumption of a relation between the quality of projects and the best practices followed is the foundation for the creation of system capable of automatic evaluation of a well-developed project, according to the community defined best practices.

This document can be divided in 7 chapters. After the introduction, Chapter 2 explores the open source world, including the differences of open source and close source development process and the need to assess the quality of this OSSP in new ways. There are numerous communities in the open source universe, but like it is shown in Chapter 3, the characteristics of Rails and the various philosophies underlying it makes this community the best starting point to understand the

[8] http://www.ubuntu.com. Ubuntu is a free & open source operating system.

role of best practices in assessing OSS. This chapter is not only about the Rails community; it provides a quick introduction to the Ruby Language[9] and the Rails Framework. In Chapter 4, the current state software metrics is exposed, the concept of Quality when addressing an OSSP, and how to measure it using classic approaches. After that, Chapter 5 discusses the concept of best practices in general and, in the particularly, the Rails community. Last but not least, to support this proposal a case-study is shown in Chapter 6: the measurement and comparison of various Rails OSSP. In the end, it was possible to create new tool to assess the quality of Rails projects and Chapter chap:conclusion describes not only what was achieved but what is still to come.

A lot of resources and the source code developed during the writing of this document is accessible at `study.gorgeouscode.com`.

[9] Ruby is an open source programming language. Ruby community is relatively young but still very focused on following best practices.

Chapter 2

Open Source

Open source describes practices in production and development where everybody has access to the product source materials. Definition from http://opensource.org/:

> *Open source is a development method for software that harnesses the power of distributed peer review and transparency of process. The promise of open source is better quality, higher reliability, more flexibility, lower cost, and an end to predatory vendor lock-in.*

Generically, open source refers to a computer software in which the source code is available free of charge for the general public to use, modify and redistribute.

Nevertheless, in the past few years, the concept of *open source* has been widely used, not just in computer software, but in every industry. Actually, new concepts such as *open design*[1] or *open religions*[2] emerged from it. A simple metaphoric example:

A restaurant would be open source if the chef reveals to the general public his cooking techniques and recipes. Consequently, by revealing his secrets, other people can start doing the same dishes and even improve his techniques.

That might not be a good bet for a restaurant, but it is proven to be a good one in software development.

In this chapter, one can read a brief history about open source philosophies and understand the role of the web open source project hosting platforms and the importance of it for the software users and developers.

2.1 The Open Source Movement

In the fifties, almost every existing software was produced by research institutes. Computers took up entire rooms at academic institutions and government agencies. No one would think that in a few decades personal computers would take the world by storm. The software was developed

[1] Open design is the development of physical products, machines and systems through use of publicly shared design information.

[2] Open-source religions attempt to employ open-source methodologies in the creation of religious belief systems.

and distributed by small communities of programmers that shared their code over private and government networks. Companies were interested in selling hardware and the free software was good advertisement for it. This was the logical way for software development.

However, with the emergence of micro-processors in the seventies, companies began to charge for software licenses. Operating systems and all kinds of software packages were seen as a product; furthermore, companies start imposing legal restrictions on software distribution and usage through copyrights, trademarks, and leasing contracts.

To fight back, in 1984, Richard Stallman created the GNU project. The goal of the project was to build a free[3] operating system.

One year later (1985), Richard Stallman created the nonprofit Free Software Foundation, with the worldwide mission to promote computer user freedom and to defend the rights of all free software users. He argues that when using free software you have four specific freedoms:

- The freedom to run the program as you wish;
- the freedom to copy the program and give it away to your friends and co-workers;
- the freedom to change the program as you wish, by having full access to source code;
- the freedom to distribute an improved version and thus help build the community.

In 1991, Linus Torvalds finished writing Linux, a unix-like kernel. Linux was not part of the GNU project, but the only missing part in the GNU project was a kernel. Linux alone was not of much help for most users. Combining Linux with the GNU system resulted in a complete operating system: the GNU/Linux system. Nowadays, there are many variants of the GNU/Linux system (often called ”distros”).

Despite the relevance of these projects for open source, it is important to note that those are not the first open source projects. In the seventies, in the University of California at Berkeley (before the GNU Project), computer science researchers were improving the UNIX system and started to build lots of applications (it became known as ”BSD UNIX”). It was 1997 when Bill Joy released Berkeley UNIX under the official moniker BSD (Berkeley Software Distribution). The copies of BSD were not completely free, but it was available to anyone who wanted it Joy only charged a small fee for it. Making the source code available to everyone enabled a world of hackers to improve on his code. Those upgrades were then filtered by him and his team for incorporation into future releases. This was the birth of a ”revolutionary” paradigm in software distribution that is now known as Open Source.

However, in the nineties, the software market was completely dominated by proprietary software from companies such as Microsoft. Even today, almost every computer sold comes with a proprietary operating system installed. Nevertheless, with the dissemination of the internet, there

[3]The word ”free” in ”free software” pertains to freedom, not price. You may or may not pay a price to get GNU software.

were new possibilities. Open source communities, sharing their software and contribute to each others, increased in size. The OSS started to gain ground from paid software.

According to NetCraft[4], the market share for top servers across the million busiest sites was 66.82% for the open source web server, Apache, much higher than the 16.87% for Microsoft web servers in May 2010. Even governments started noticing open source during the last few years, and in some case adopted it [Hah02]. The broad acceptance of OSS means that now OSS is not only used by computer specialists.

John Powell[5] has declared that measuring the savings that people are making in license fees, the open-source industry is worth 60 billion dollars. Matt Asay[6] shares the view that from the customers perspective open source can be now considered the largest software industry in the world. The full review can be found at CNET News[7].

As seen here, open source projects follow a series of principles of freedom. It is not as simple as cost-free software. In fact, there is nothing in the open source licenses preventing people from taking a previously free OSS and charging for it, but because every one can redistribute it without charging, it wouldn't make any sense. To tell the truth, by changing the business models and being more service oriented, it is possible to create lucrative businesses around open source. Canonical[8] is nice example of this because their business model is to provide technical support and professional services related to Ubuntu[9]. Ten core principles about open source software can be found at Ubuntu web site:

- Software must be free to redistribute.
- The program must include source code.
- The license must allow people to experiment with and redistribute modifications.
- Users have a right to know who is responsible for the software they are using.
- There should be no discrimination against any person or group.
- The licence must not restrict anyone from making use of the program in a specific field.
- No-one should need to acquire an additional license to use or redistribute the program.
- The license must not be specific to a product.
- The license must not restrict other software.

[4] http://news.netcraft.com/archives/2010/05/14/may_2010_web_server_survey.html/, accessed on 2010/12/21.
[5] John Powell is CEO, President, and Co-founder, Alfresco Software Inc.
[6] Matt Asay is chief operating officer at Canonical, the company behind the Ubuntu Linux operating system.
[7] http://news.cnet.com/8301-13505_3-9944923-16.html/ accessed on 2010/12/21.
[8] Canonical Ltd. is a private company that created Ubuntu. All started on 8 July 2005, when Mark Shuttleworth and Canonical Ltd. announced the creation of the Ubuntu Foundation and provided an initial funding of US$10 million.
[9] Ubuntu is a free & open source operating system.http://www.ubuntu.com.

- The license must be technology-neutral.

In the end, for many people open source is a philosophy of life. But regardless one's beliefs, it is a fact that open source software development, worked quite well for many new and already established companies during the last years.

2.2 Open Source Software Development

Usually, large industries have a strict organization model. This is not the way open source communities operate. Open Source communities work in what can be called *bazaar style*. This term was introduced by Raymond [RE00]. He compares the traditional software development process to building cathedrals: a few specialized individuals working in isolation and every one is told exactly what they should do. Open source development seemed to resemble a great babbling *bazaar*, where everyone can be part of the project, contribute to, and change it. Because of this nature, the specification of an open source software project evolves in an organic way [CM07].

Of course, OSS is not developed in the same way all the time. Each community has particular habits. Different development and management methodologies, such as more traditional or more agile, can be used. Currently, the truth is that the most successful communities organize themselves in a similar way as professional and proprietary companies, and some of the big open source projects have big companies supporting them, without being a charity. Imagine the consequences of a having a handful of highly motivated eyes going through the code, constantly reviewing it, correcting, and adding to it. These are people working not because they were told to, but because it is their own will. The communities are the strength of OSS and the companies behind it.

OS development makes possible to a project to reach a high quality level, in much less time and with fewer financial investments, comparing to traditional software development. Nevertheless, these OSP still follow the OS core rules, and those projects are community driven. The users and developers must feel engaged to it.

It is obvious that companies need to make money, but even if their software is free and open source, new ways of income can be explored. For example, charge for support or related services, donations, etc.

The well known open source browser, Firefox, is the descendant of the graphical web browser named Mosaic released by Netscape in 1993. When Microsoft bundled Internet Explorer with Windows, it was obvious that Netscape was doomed. However, they turned project into open source, created the Mozilla Foundation, and the community gathered around it, helping the company regain the lost market.

This and other examples shows OS develoment as one of the most effective development models today. Many companies are trying to explore these business models.

2.3 Open Source Project Hosting Platforms

The strength of the open source development model comes from the user base and the power given to it. Users should fill out bug reports, submit feature requests, etc. Because the developers can be spread all around the globe, there is the need of effective administered communication channels for better cooperation and co-ordination.

An open source project hosting platform is the central tool that supports and coordinates the development of an open source project. Normally it is in a form of a website.

Since 1999 (year that SourceForge was launched), many open source project hosting websites (OSPHWs) were created to host open source projects. OSPHWs offer different features, like codebase [10] hosting (a project codebase is typically stored in a source control repository), code review, bug tracking, web hosting, wiki, mailing list, etc [BHK06].

Table 2.1 shows a list of the most used OSPHW. By looking at this table we can see that SourceForge is the best established OSPHW. It is also one of the eldest, hosts more than 230,000 projects, and has more than 2 million registered users [CM05].

GitHub is one of the youngest OSPHW (launched in 2008). However, in only two years, it drew more than 500,000 users (one quarter of sourceforge users) and is hosting more than 1,500,000 projects. The only version control system provided by GitHub is Git[11]. Because GitHub projects are in fact Git repositories, it is incredibly easy to make branches and merges in GitHub. Although branching was considered a big pain in older version control systems, it turned out that by using Git, it can in fact improve the developers' collaboration and organization. This happens because of the distributed philosophy and implementation of Git Hosting a Git repository is not difficult, but coordinating efforts of forking and merging amongst people is tough. With a system like Github, it becomes a lot easier [Coo86]. However, the main reason for GitHub popularity is the social aspect of it. Users and projects have public profiles and activity feeds which display activity on public projects such as commits, comments, forks, etc. Furthermore, with so many high profile projects on board (jQuery, reddit, Sparkle, curl, Ruby on Rails, node.js, ClickToFlash, Erlang/OTP, CakePHP, Redis), it is easy to imagine that GitHub could be the next SourceForge.

The reasons above allow us to believe that GitHub has a strong and growing open source community, and that it is an important platform both for users and developers. Because of that, and the high number of Ruby on Rails projects hosted here, it was decided to use GitHub projects for the studies shown in later chapters.

[10] The term codebase means the whole collection of source code used to build a particular application or component.
[11] http://git-scm.com/. Git is a free & open source, distributed version control system that Linus Torvalds developed to help manage Linux kernel development.

Open Source Project Hosting Web Sites					
Name	Established	Available VCS	Users	Projects	Alexa rank [a]
SourceForge	1999	CVS SVN Bazar GIT Mercurial	2,000,000	236,319	136
GitHub	2008	GIT	505,000	1,516,000	742
Google Code	2006	SVN Mercurial	?	250,000	900[b]
Code Plex	2006	SVN Microsoft TFS Mercurial	151,782	15.955	2,343
Assembla	2006	SVN GIT	180,000	60,000	6,628
Launchpad	2005	Bazar	1,140,345	19,016	12,466
BerliOS	2000	CVS SVN GIT Mercurial	47,285	5,448	17,299
Bitbucket	2008	Mercurial	51,600	27,769	12,047
Gitorious	2008	GIT	?	8,336	28,531
GNU Savannah	2000	CVS SVN Bazar Arch GIT Mercurial	48,593	3,233	48,286

Data retrieved on 2010-12-20, from each of the OSPH Websites, and using Alexa rank website.

[a] Alexa rank represents the approximate number of websites, in the world, that have a higher popularity than the given site (the smaller the better).

[b] This value is an approximation.

Table 2.1: Open source project hosting websites

2.4 The Need to Assess Open Source Projects

It was said before that OSS development resembles a babbling *bazaar*. Therefore, can software that is developed in such chaotic way be trusted as a high quality product?

The shock is that in fact the *bazaar style* seemed to work [HS02].

Some big projects, for instance Linux distributions such as Ubuntu[12], are proof. However, most projects are not that big. Small communities can start and maintain projects to solve their common

[12] http://www.ubuntu.com. Ubuntu is a free & open source operating system.

problems. Actually, anyone can start an OSP, but sometimes it is hard to distinguish the project developers from the project customers/users. Because of this, bug reports and wanted features can become indistinguishable too. The specification of an open source software project evolves in an organic way [CM07].

Due to the fact that nowadays the best place the find open source projects are platforms like GitHub, it seems that these platforms should give information about the quality of the projects. In fact, GitHub has already done some work regarding to automatic code analysis of hosted projects. But at the time of writing, it only shows a fews graphs based on simple metrics, such as number of commits by each contributor and number of programing languages used. It enables the user to understand how the developers are involved in the project, the programming languages most used, and whether the project is active or not. However, there exists nothing that can quickly inform the user about the quality of the project source.

We believe that there is a need for a more advanced analysis report. Both developers and users need to assess OSSP quality. It enable users to make better choices and developers to further improve their software.

Chapter 3

Ruby on Rails

The web application framework[1] Ruby on Rails (Rails) is an excellent example of a successful open source project. However, we cannot discuss Rails without first discussing the Ruby programming language.

It is a common misconception to believe Ruby and Ruby on Rails are the same thing. Although closely related, they are distinct. When someone says Ruby, they are referring to a programming language. Ruby on Rails refers to a full-stack framework to develop web applications that was created using Ruby. Rails helped propel movements like Test Driven Development, Pair Programming and other Agile Methodologies. It also played a major role in Ruby adoption.

In this chapter, one can read a brief history about the Ruby language and learn its most distinctive characteristics. Next, find out how Rails turned to be one of the most recognized frameworks for developing web applications. After discussing the main concepts involved in the Rails framework, a section is dedicated to the framework structure. The chapter ends with a characterization of the Rails community.

3.1 Ruby Programing Language

Ruby is an open source, dynamic and object oriented programming language. It is not only free of charge, but is also free to use, copy, modify, and distribute. Ruby was created by Yukihiro Matsumoto (also known as *Matz*) and publicly released in 1995. It achieved mass acceptance in 2006.

Matsumoto blended parts of his favorite languages (Perl, Smalltalk, Eiffel, Ada, and Lisp), balancing functional programming with imperative programming to create a multi-paradigm language. The objective was to create a *natural* language. This does not mean that Ruby is simple. Matsumoto often explains the difference by saying:

Ruby is simple in appearance, but is very complex inside, just like our human body.

[1] A web application framework aims to minimize the overhead associated with common activities performed in Web development. For example, it might provide libraries for database access, templating frameworks or session management. It often promotes code reuse.

He believes people want to express themselves when they program. Programmers do not want to fight with the language. Programming languages must feel natural to them. Ruby is based on ”Principle of Least Surprise”, which means the language behaves the way you expect it to behave.

In Ruby, everything is an object. Even basic datatypes, such as numbers or booleans. Furthermore, every operations in an object is a method, and every method returns an object. A basic rule of the object oriented paradigm is that every object has a class. So if you call the method class on any object, it will return another object representing the first object class; since this last object is also an object, it will respond to the same method. A class object will respond to the method class returning the class object ”Class” whose class is itself. The class ”Class” can be considered a metaclass, because its instances are classes, too. In 3.1 a simple snippet of Ruby code shows this tricky characteristic. The code can be run in IRB[2]. This tool is of great help for Ruby programmers, and an easy way for newcomers to start learning.

Listing 3.1: Sample Ruby Code

```
>> # You can call methods on directly on numbers because they are
   objects.
>> 1.to_s
=> "1"

>> # Everything is an object, and every object responds to the
   class method.
>> "1".class
=> String

>> # Even classes are seen as objects.
>> "1".class.class
=> Class

>> # Class is a metaclass.
>> "1".class.class.class
=> Class
```

Ruby is dynamic and interpreted at run time with duck typing[3] ”If it walks like a duck and quacks like a duck, then it is a duck!”.

Ruby has open classes, which means that at any moment, it is possible to change any class

[2]Interactive Ruby Shell (IRB) is a shell for programming in the object-oriented scripting language Ruby. It can be launched from a command line and allows the execution of Ruby commands with immediate response, experimenting in real-time.

[3]Duck typing, in object oriented programing languages, is a style of dynamic typing in which the object methods and properties determine the valid semantics, rather than its inheritance from a particular class or implementation of a specific interface.

methods and definitions. This also means that programmers are allowed to change predefined classes. The truth is that depending on the usage, it can be either a good or a bad thing. Carrying out some of those things clearly violates the object-oriented principle of encapsulation. An example of a bad usage would be to monkey patch[4] the size or length method of string objects to make it ignore white spaces. It might be useful for certain situations, but it might also give a big headache to someone coming to the project a few days later.

However, if it were not for those characteristics, Rails would not have been able to easily change some of Ruby basic datatypes, and consequently developers would not be able to write amazingly readable code, as shown in 3.2.

Listing 3.2: Sample Ruby Code (Rails Environment)

```
>> # Calling the method \"today\" on the class \"Date\"
>> # returns a date object.
>> Date.today
=> Sat, 11 Feb 2012

>> # It is possible to do calculations on those objects,
>> # rails extends the Fixnum class with nice methods to
>> # help on those calculations
>> Date.today + 3.months - 7.day
=> Fri, 04 May 2012
```

Some of these Ruby characteristics are seen by some people as amazing features, and many careful programmers call it ”dark magic”. In the end, it always comes to the good sense of the people writing the code. There is no doubt that the Ruby language has drawn thousands of devoted coders worldwide. The reason behind that can only be due to its elegant syntax that is natural to read and easy to write. In addition, many of the philosophies that are present in the Ruby on Rails framework, such as ”Don’t Repeat Yourself” and ”Principle of Least Surprise” (we will talk in more detail about that later in this chapter), are also shared by Ruby language. Ruby programs are easy to read and understand, even for a newcomer. All this makes Ruby projects highly maintainable, and the programmers using it feel at ease. As its inventor Matsumoto concluded in his presentation at the ACM Finals in Japan of 2007:

Usability matters, feeling matters. Don’t think, feel.

[4]The term monkey patch means any dynamic modification to a class and is often used as a synonym for dynamically modifying any class at runtime.

3.2 Ruby on Rails Framework

In 2001, David Heinemeier Hansson was hired by Jason Fried to build a web-based project management tool, which ultimately became the 37signals[5] Software as a Service[6] product named Basecamp[7].

He decided to start the project by developing a custom web framework using the Ruby programming language to avoid what he saw as repetitive coding inherent in platforms such as Java. Ruby was almost unknown at the time.

The framework he created was later released as an open source project, separate from the project management tool. The name of this open source web framework is Ruby on Rails.

In 2005, Hansson called the attention of the community with a legendary video named "Creating a Weblog in 15 minutes." It was an introduction on development with Rails. In the same year, his creation earned him the Google-O'Reilly Best Hacker award. Moreover, the success of Ruby on Rails is considered the most responsible for Ruby's mass acceptance.

More Ruby frameworks were born along the away, Merb[8] being one of them.

Nevertheless, on 23th of December, 2008, it was announced (by David in the Rails web site)[9] that Merb would be merged into Rails 3. This ended unnecessary duplication on both open source communities, and closed the debate about when to choose one over the other. The best ideas, from both sides of the fence, were chosen to create a better and stronger project. This a nice example of how the open source Ruby community works.

This is a decent advantage in comparison with, for instance, the Java world, where you can find dozens of different frameworks for the same purposes. Because there are so many choices, developers end up confused. Developers are part of different communities. It seems that when they are using different languages, most of the time they cannot try the different options and they do not know which one is better. It is just a matter of belief (like a football team). The problem is that these subgroups might have a common goal, but are paddling in different directions. Due to this, there is less collaboration and the progress is slower. Although, with an incredible exponential grow in the last years, which might lead to different and new ways of thinking, this big division does not seem to be happening in the Rails community.

The Rails philosophy includes several guiding principles:

- Model-View-Controller (MVC): Rails uses the MVC architecture and is intended to be used with an Agile development methodology, providing developers a rapid Web application development environment.

[5] 37signals is a privately held web application company based in Chicago, Illinois. The firm was co-founded in 1999 by Jason Fried, Carlos Segura, and Ernest Kim.

[6] Software as a service (SaaS) is a software delivery model in which software and associated data are centrally hosted on the cloud. Typically, in SaaS, users access software using a thin client via a web browser.

[7] Basecamp is a web-based project-management tool developed by 37signals and launched in 2004.

[8] Like Ruby on Rails, Merb is an MVC framework built using ruby. Unlike Rails 2 (a big framework), Merb adopted an approach that focused on essential core functionality, it was built for speed, leaving extra functionalities to plugins.

[9] http://weblog.rubyonrails.org/2008/12/23/merb-gets-merged-into-rails-3

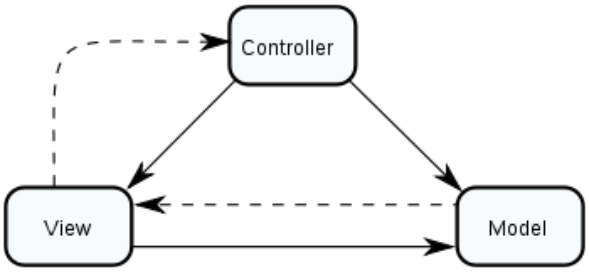
- Convention over Configuration: Rails makes assumptions about what you want to do and how you are going to do it, rather than letting you tweak every little thing through endless configuration files.
- Don't Repeat Yourself (DRY): Writing the same code over and over again is a bad thing. DRY is a principle of software development aimed at reducing repetition of information of all kinds.
- REST (Representational State Transfer) - A pattern for Web application, organizing your application around resources and standard HTTP verb is the fastest way to go.

These will be described in detail in following sections.

3.3 Model-View-Controller

MVC is an architectural pattern used in software engineering. Successful use of the pattern isolates business logic from user interface considerations, resulting in an application where it is easier to modify either the visual appearance of the application, or the underlying business rules, without affecting the other. MVC was first described in 1979 by Trygye Reenskaug.

Figure 3.1: Model-View-Controller Diagram



- A *model* represents the information of the application and the rules to manipulate that data. In Rails, models are used for managing the rules of interaction with a corresponding database table. In most cases, one table in your database will correspond to one model in your application. Your application logic will be concentrated in the models.
- A *view* represents the user interface of your application. In Rails, views are often HTML files with embedded Ruby code (which we call erb templates) to perform tasks related to the presentation of the data. Views handle the job of providing data to the web browser or other tool that is used to make requests to your application.
- *Controllers* provide the "glue" between model and views. In Rails, controllers are responsible for processing the incoming requests from the web browser, interrogating the models for data, and passing the data to the views for presentation.

The benefits of using MVC include the isolation of business logic from the user interface. In addition it becomes easier to keep the code DRY, and it makes clear where different types of code belong for easier maintenance.

3.4 Convention over Configuration

Traditionally, frameworks need multiple configuration files, each with many settings. These provide information specific to each project, ranging from URLs to mappings between classes and database tables. With the complexity of an application, the size and number of those files grows as well. Most of the time, it is very hard to maintain a lot of configurations files. Rails was developed to minimize these issues by following the *Convention over Configuration* paradigm.

Convention over Configuration aims at simplifying the development without losing the application flexibility. It means you do not need to write configuration files in order to have a flexible application. This leads to less code and less repetition. The Rails creator calls this "Intelligent Patterns". If you do not want to configure anything, just follow the conventions and the framework will know what to do.

To better understand *Convention over Configuration*, let's see how Rails analyses a URL such as the following:

Listing 3.3: URL Example

```
/account/show/1
```

By default, the framework will split the URL by "/" and following the rest standards, we can expect a few things to be true:

- *account* – It probably means that a controller called "AccountController" exists in the application. It should be a class that extends the ApplicationController class and handles all kind of actions related with accounts.
- *show* – The AccountController class should have a show method. This method will probably interact with a model called Account to fetch the needed data and pass it to a view, which will render a nice page showing the account information.
- *1* – Since we are in the accounts show action, this is the parameter called "id" with the value "1". It means that user want to see information about the account whose id is 1.

Although it is fairly easy to change this behavior (in fact, it is a best practice to change it a little bit) without writing one line of code, a Rails application freshly created will expect all those things. We simply need to create the controller, a view, and have a database, with the correct names and everything will work by itself. In many others frameworks, it is necessary to create one or more configuration files, normally XML files.

Another example of *Convention over Configuration* is with respect to the persistence layer (which typically deals with a database). The only thing you need to do in order to map a Model to its table in the database is the code shown in 3.4.

Listing 3.4: Sample Model file "product.rb"

```
class Product < ActiveRecord::Base
end
```

That is enough for the class to be bound by the framework with a table in the database called Products, and all of its columns will be accessible for use without creating a configuration file to map it. Note as well that you do not need to create getter and setter methods, as they will be there ready to use. Rails has a concept of pluralize, which means a model Product will have a table Products in the database, or a model Customer will have a table Customers, and so on.

3.5 Don't Repeat Yourself

Don't Repeat Yourself (DRY) is an approach aimed at reducing duplication. The philosophy emphasizes that information should not be duplicated, because duplicates increase the difficulty of code maintenance, decrease clarity, and lead to opportunities for inconsistencies.

DRY is applied quite broadly to include database schemas, test plans, the build system, and even documentation. When the DRY principle is applied successfully, a modification of any single element of a system does not change other logically unrelated elements. DRY code is created by data transformation, which allows the software developer to avoid copy and paste operations. DRY code usually makes large software system easier to maintain, as long as the data transformations are easy to create and maintain. DRY is not about just avoiding code duplication, but more generally about avoiding multiple and possibly diverging ways to express every piece of knowledge: e.g., logic, database schemas, and constants. If we are always repeating the same code, refactoring will be in order to keep your code DRY compliant.

3.6 The Framework Structure

Every Rails application follows the same file structure organization. This way everything should be in the right spot.

After generating a new Rails application (this is achieved by running just the single command line shown in Listing 3.5)

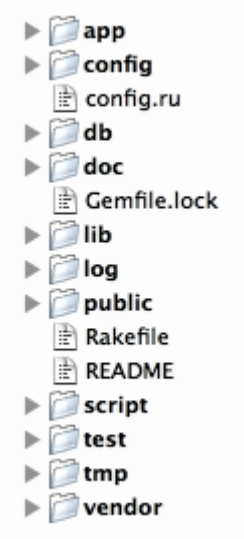
Listing 3.5: Command to create a new rails application

```
rails new <name_of_the_appliction>,
```

a folder looking like the tree in Figure 3.2 should be created. It is a ready to run web application.

The folder called *app* is where the majority of the real code for the new application goes (models, views, controllers, helpers, etc). The other folders are usually destined to documentation, configuration, third party code, temporary files, etc.

Figure 3.2: Rails Project File Structure



Rails is a meta-framework[10], it is composed of the following frameworks:

3.6.1 Active Record

Active Record connects business objects and database tables to create a persistable domain model where logic and data are presented in on wrapping. It's an implementation of the object-relational mapping (ORM) pattern by the same name as described by Martin Fowler. Active Record is the base for the models in a Rails application.

It provides database independence, basic CRUD (Create, Read, Update, and Delete) functionality, advanced finding capabilities, and the ability to relate models to one another, among other services.

3.6.2 Action Pack

Action Pack splits the response to a web request into a controller part (performing the logic) and a view part (rendering a template). This two-step approach is known as an action, which will normally create, read, update, or delete some entity defined by a model (often backed by a database) before choosing either to render a template or redirecting to another action.

Action Pack implements these actions as public methods on Action Controllers and uses Action Views to implement the template rendering. Action Controllers are then responsible for handling

[10] A meta-framework is a framework composed of smaller frameworks.

all the action relating to a certain part of an application, such as listing, creating, deleting, and updating records.

Action View templates are written using embedded Ruby in tags mingled in with the HTML. To avoid cluttering the templates with code, a bunch of help classes provide common behavior for forms, dates, and strings. Additionally, it is easy to add specific helpers to keep the separations as the application evolves.

3.6.3 Action Mailer

Action Mailer is a framework for building e-mail services. You can use Action Mailer to send emails based on flexible templates, or to receive and process incoming email.

3.6.4 Active Support

Active Support is an extensive collection of utility classes and standard Ruby library extensions that are used in the Rails. All these additions have hence been collected in this bundle as a way to gather all that sugar that makes Ruby sweeter. For instance, the examples shown previously regarding date and time calculations are part of this bundle.

3.7 Rails Community

The biggest part of Ruby developers are in fact Rails developers. Moreover, Rails is considered the biggest influence on Ruby popularity (tens of thousands of Rails applications are online). In addition, Rails developers are also web developers, so we can assume that the majority of Ruby and Rails developers also write HTML, Javascript and CSS. The Ruby and Ruby on Rails community has been growing up in the last few years. Programmers from other languages such as Java and .NET are discovering the power of Ruby and how easy it is to create web application using Rails.

Ruby and Ruby on Rails community members love conventions and best practices. It is also common to associate Ruby on Rails with Behaviour Driven Development (BDD) and Agile methodologies. For instance, a lot of Ruby on Rails book authors speak about automated tests, written using domain specific languages (DSLs) such as Cucumber or Rspec. This may seem to be an advanced topic, but writing automated tests in Ruby is fairly easy, and DSLs like Cucumber and Rspec follow the human/natural philosophies of Ruby.

In 3.6, a simple Cucumber test file shows how readable Cucumber tests can be.

It looks like a use-case, but this is a real automated test. By using the power of regular expressions and Ruby dynamic capabilities, it is possible to parse this Cucumber file and check if the software is valid according to the specification.

Listing 3.6: Cumber feature

```
Feature: Search courses
  Potential students should be able to search for courses

  Scenario: Search by topic
    Given there are 34 courses which do not have the topic "math"
    And there are 2 courses LEI, MEI that each have "math" as one of
       the topics
    When I search for "math"
    Then I should see the following courses:
      | Course code |
      | LEI         |
      | MEI         |
```

There are a lot of open source projects, frameworks and communities. However, as it was shown throughout this chapter, the characteristics of Rails and the various philosophies underlying it makes this community of great potential to be a starting point to understand the role of best practices, their benefits, and how to measure it.

Later in this document, we will identify which ones are the most relevant Rails best practices, and discover if the projects that do follow best practices are truly the projects with more success in the community.

Chapter 4

Software Metrics

In this chapter, you will learn the basics of software metrics. A brief explanation of quality attributes is given. Furthermore, we stand that, in the context of open source software, maintainability is the most relevant attribute. A few traditional software metrics are listed.

4.1 Assessing Open Source Software

The simplest operation in science and the lowest level of measurement is classification [Kan02].

By assessing OSS we mean to sort OSS projects into an ordinal scale[1] This can be achieved by defining a ranking system[2] and by placing OSS projects into quality categories with respect to certain quality attributes. First, we need to find a way of quantifying those OSS quality attributes.

In software, quality is an abstract concept. It is commonly recognized as lack of ”bugs,” and meeting the functional requirements. However, quality can be perceived and interpreted differently based on the actual context, objectives, and interests of each project. Many software development companies do monitor costumer satisfaction as a quality index. For instance, IBM ranks their software products in levels of CUPRIMDSO [Kan02]:

- Capability/Functionality (refers to the software meeting its functional requirements)
- Usability (refers to the required effort to learn and operate the software)
- Performance/Efficiency (refers to the software performance and resource consumption)
- Reliability (refers to software fault tolerance and recoverability)
- Instalability/Portability (refers to the required effort to install or transfer the software to another environment)
- Maintainability (refers to the required effort to modify the software)

[1] Ordinal scale refers to the measurement operations through which the subjects can be compared in order.

[2] Ranking system example: to classify a quality attribute, for instance the project documentation, according to its quality with five, four, three, two or one star.

- Documentation/Information (refers to the coverage and accessibility of the software documentation)
- Service (refers to the company monitoring and service)
- Overall (refers to an overall classification based on the other attributes)

Almost every big software company has similar quality attributes. ISO/IEC 9126 provides a framework for the evaluation of software quality (The goal is to achieve quality in use, in other words, quality from the user perspective) [Bev99] IISO/IEC 912 defines six software quality attributes:

- Functionality (refers to the software meeting the functional requirements)
- Reliability (refers to software fault tolerance and recoverability)
- Usability (refers to the required effort to learn and operate the software)
- Efficiency (refers to the software performance and resource consumption)
- Maintainability (refers to the required effort to modify the software)
- Portability (refers to the required effort to transfer the software to another environment)

Quality attributes have interrelationships. They can be conflictive[3] or support[4] one another. For example, the higher the functional complexity of the software, the harder it becomes to achieve maintainability [Kan02].

Because of the OSP bazaar style and continuous development process, it is intuitive that the maintainability and documentation attributes have a big influence on the overall quality and continuous progress of an OSP. Maintainability and documentation have support relationships with usability, reliability and availability attributes, but might be conflictive with functionality and performance attributes.

Failure to meet functionality often leads to late changes and increased costs in the development process. The software industry and researchers have been mostly interested on testing methodologies that focus on functional requirements and pay little attention to non-functional requirements [CdPL09].

There are several challenges and difficulties in assessing non-functional quality attributes for software projects. For example, security is a non-functional requirement that needs to be addressed in every software project. Therefore, badly-written software may be functional, but subject to buffer overflow attacks. Another example is the amount of codebase comments. If the code does not have any comments, it will not affect the functional requirements, but it is obvious that it will decrease readability and maintainability [GKS+07].

[3] Conflictive, negative influence, if one attribute is high it makes the other one low.
[4] Support, positive influence, if one attribute is high it makes the other one high too.

4.2 Classic Software Metrics

To classify OSS with regards to a certain quality attribute, we need to find which factors influence it. Then we need a way to measure that attribute. If we need to make measurements, we need metrics.

Fortunately, there are around two thousand documented software metrics, but there is few information on how those metrics relate to each other. Most of them simply have different names, but give similar information [FN99]. The major challenge is to discover how important the information given by those metrics is, if the calculation effort pays off, how to interpret their values, and find correlations[5] to assess the quality attributes of an OSP.

4.2.1 Lines of Code

A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements in the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements. [CDS86] The lines of code (LOC) metric is anything but simple. The major problem comes from the ambiguity of the operational definition: the actual counting. In the early days of Assembler programming, in which one physical line was the same as one instruction, the LOC definition was clear. With the availability of high-level languages, the one-to-one correspondence broke down. Differences between physical lines and instruction statements (or logical lines of code), and differences among languages contribute to the huge variations in counting LOC. For instance, Boehm [Boe09] counts lines as physical lines and includes executable lines, data definitions, and comments. Even within the same language, the methods and algorithms used by different counting tools can cause significant differences in the final counts.

Next, the most common variations are described [JJ86]:

- Count only executable lines.
- Count executable lines plus data definitions.
- Count executable lines, data definitions, and comments.
- Count executable lines, data definitions, comments, and job control language.
- Count lines as physical lines on an input screen.
- Count lines as terminated by logical delimiters.

[5] Correlation is probably the most widely used statistical method to assess relationships among observational data [Kan02].

4.2.2 Cyclomatic Complexity

The measurement of cyclomatic complexity [McC76] was designed to indicate a program's testability and maintainability. It is the classical graph theory cyclomatic number that indicates the number of regions in a graph. As applied to software, it directly measures the number of linearly independent paths through a program source code. Basically, it counts how many different executions a program can have. For instance, one "if" statement will double the number of paths. Therefore, a high number of control statements (ifs, loops, etc) in a program source code will result in a high cyclomatic complexity. As such, it can be used to indicate the effort required to test a program. To determine the paths, the program procedure is represented as a strongly connected graph with unique entry and exit points. The general formula to compute cyclomatic complexity is:

$$M = V(g) = e - n + 2p$$

where

- $V(g)$ is the cyclomatic number of g
- e is the number of edges
- n is the number of nodes
- p is the number of unconnected parts of the graph

4.2.3 Fan-In and Fan-Out

Fan-in and fan-out are perhaps the most common design structure metrics, which are based on the ideas of coupling [YC79]:

- *Fan-in* is a count of the modules that call a given module
- *Fan-out* is a count of modules that are called by a given module

In general, modules with a large fan-in are relatively small and simple, and are usually located at the lower layers of the design structure. In contrast, modules that are large and complex are likely to have a small fan-in. There is also the theory that high fan-outs represent a high number of method calls and thus are undesirable, while high fan-ins represent a high level of reuse [WLCR07].

4.2.4 Object-Oriented Metrics

Classes and methods are the basic constructs of OO technology. The amount of function provided by an OO software can be estimated based on the number of identified classes and methods or

their variants. Therefore, it is natural that the basic OO metrics are related to classes, methods, and their size.

The pertinent question therefore, is what the optimum value should be for OO metrics. There may not be one correct answer, but based on his experience in OO software development, Lorenz proposed eleven metrics as OO design metrics called rules of thumb [LK94].

- *Average Method Size (LOC)*: Should be less than 8 LOC for Smalltalk and 24 LOC for C++
- *Average Number of Methods per Class*: Should be less than 20. Larger averages indicate too much responsibility in too few classes.
- *Average Number of Instance Variables per Class*: Should be less than 6. More instance variables indicate that one class is doing more than it should.
- *Class Hierarchy Nesting Level (Depth of Inheritance Tree, DIT)*: Should be less than 6, starting from the framework classes or the root class.
- *Number of Class/Class Relationships in Each Subsystem*: Should be relatively high. This item relates to high cohesion of classes in the same subsystem. If one or more classes in a subsystem don't interact with many of the other classes, they might be better placed in another subsystem.
- *Average Number of Comment Lines (per Method)*: Should be greater than 1.

4.3 Available Tools

Tools are intended to make a task easier. Codebase analysis tools are designed turn the task of analysis and measure source code easier. Those tools help in the process of finding software flaws and can also serve as aids for assessing the quality of software.

Next a list of some free tools:

- FindBugs finds "bugs" in Java programs
- FxCop (Microsoft) analyzes managed code assemblies and reports information about the assemblies
- PMD scans Java source code and looks for potential code problems
- PreFast (Microsoft) is a static analysis tool that identifies defects in C/C++ programs
- RATS (Fortify) scans C, C++, Perl, PHP and Python source code for security problems like buffer overflows and Time Of Check, Time Of Use race conditions
- SWAAT simplistic tool for Java, JSP, ASP .Net, and PHP

- Flawfinder scans C and C++
- Saikuro is a Ruby cyclomatic complexity analyzer.
- Reek is a tool that examines Ruby classes, modules, methods and reports any code smells it finds.
- Rails best practices is another tool that examines Ruby classes, modules, methods and reports any code smells it finds.

Tools like the ones listed above analyze code without executing it and point out what they consider to be potential weaknesses. The most typical example of what those tools can find is most likely calls to the emphgets function in the C programming language. This function is inherently insecure and can lead to buffer overflows. Specially crafted user input values can, for instance, allow an attacker to access or modify confidential data or even take control of any computer executing that piece of software.

Because these tools need to ”understand” the code being analyzed, they are necessarily very language specific. Furthermore, when analyzing a large amount of software projects, for instance GitHub projects, it is of great importance to have a previous knowledge of what kinds of projects are going to be found (programming languages, frameworks, and other attributes). This way tools can be adapted to get better results.

Chapter 5

Best Practices

Best practices are not standards. Instead, they are recommendations followed by a specific community. It is something that most people agree with, although it might not be scientifically proven, and seems to be the best way to handle a problem.

In this chapter we will understand how important best practices are for open source communities, how they give meaning to traditional software metrics, and identify some examples of best practices from the Ruby community.

5.1 Best Practices in Open Source Software Projects

Open source communities have a tendency to create *coding rules*, i.e., *principles governing the conduct of programmers and serving as a basis of measure or judgment*. It is a natural and evolutive process for people surviving in an open space. We can call these natural rules *best practices*.

Best practices are methods thought of as being the best way to achieve something; they are spread through the community and everybody follows that way. It is obvious that when a developer follows well established principles and best practices, the project maintainability is increased. Consequently, project newcomers will find it easier to understand the project code [Dro02]. In addition, there is more than just that, and following best practices discourage:

- Poor performance (due to bad patterns)
- Poor error checking (defensive programming)
- Inconsistent exception handling / Maintainability (long-term quality)

To attain the same benefits, companies define standards. This is something considered by an authority as a basis of comparison and a normal requirement for quality. By other words, it is an approved behavior model for their workers. However, these principles are defined by the few people on top and then spread down on a pyramid. Many times, those rules are not well thought-out by the project leaders, and this can block the progress.

In the other hand, the apparent chaos of open source also requires some rules, but contrary to the companies' coding standards, which work in a top down way, best practices happen in a bottom up and distributive mode. Everybody can try different ways of doing things, but the ones with better results are most likely to be copied.

A simple metaphor exposes the difference:

Companies use traffic lights where open source communities use roundabouts.

Both, the strict company standards (traffic lights) and the OSP best practices (roundabouts) are ways to regulate intersections. The result of traffic lights is easier to predict, however that regulation system does not depend much on the drivers skills. Because it is so restrictive, often found is a driver stooped alone at the crossroads waiting for a green light, losing precious time. On the other hand, the roundabout system is a less restricted system and relies much more on the quality of the drivers, but it opens the possibility to a much more efficient way to avoid a traffic jam.

There is little work done concerned with measuring coding best practices by automatic analysis of source code. A plausible explanation for this is the fact that best practices are not a set of immutable rules; they are a continuous evolution and improvement of developmental methodologies. Communities are constantly creating rules and best practices, even without noticing it. It is not possible to write down a list of best practices without some ambiguities.

At first glance, best practices metrics seem to be for classic metrics as natural as medicine is for science. This is not the case. In fact, classic metrics, on their own, do not give much information about a project. In many cases, best practices can be the key to understanding what the optimum value for a classic metric should be, like, for instance, determining *how many lines of code should a ruby method have*.

Of course, those questions are subjective. However, by analyzing renowned projects, developers' opinions, and so on, one can find a best practice that gives a plausible answer to the search for the *most favorable value*.

In addition, one can use known source code metrics and, by analyzing their values, to find new correlations that might give hints about whether some methodological approaches were taken into account during the project development process.

We believe that best practices can give a meaning to metrics.

5.2 Identifying Best Practices

We understood that software projects can benefit extensively from using best practices. However, what is a best practice after all? The truth is that everything can be a best practice, for example: the use of two spaces to indent code and no tabs, writing unit tests for your code, the way files are organized inside a project, etc.

Some of those things might look like a matter of taste, but the truth is that every worthy Ruby developer uses two spaces to indent code. This is the standard for the Ruby community. Other communities, for instance JavaScript programmers, prefer 4 spaces, and in the Java world 8 spaces is considered to be a good choice.

It is important to note that we can find virtually no Ruby programmer using a different indentation, but we can find Java communities advocating different indentations. This fact shows that the Ruby community agrees that 2 spaces is a the best option and can be considered a best practice. In contrast, we can not be completely sure about the 8 spaces for Java, since there are also a lot of developers advocating 4 spaces, and also a good number using tabs instead. The Java community is divided, and it is almost possible to identify sub-communities defending different answers for the same questions. It might be possible to identify conventions in those smaller communities, but it will be more difficult to indentify them for the largest community.

To consider these conventions as a best practice, it is important to understand whether those conventions are strong throughout the community in question. Because of that, it is obvious that the first step before finding best practices is to identify the community: are we trying to find best practices for Ruby programmers? For all programmers in general? For the programmers working for a specific company or project?

In smaller communities, it might be easier to achieve agreement and consequently find best practices. However, best practices created and followed by a small groups are less likely to be considered strong best practices compared to best practices followed by the developers working on the top 10 open source projects.

It is clear that best practices are specific to a certain community; so, after correctly choosing a community, to identify best practices we need to find out what patterns are used by its members, and to prove that they are real best practices that should be followed it is important to prove that some benefit comes from using it.

The most obvious benefit from using it is that the project maintainability is increased. For example, people in the community might be expecting to find the code indented with two spaces or methods named in camelcase, constants in upcase, etc.

Apparently, using two or more spaces does not directly affect the code quality (other than maintainability), but it is reasonable to infer that a developer is new to Ruby if he does not know it. In other words, following or not, best practices has a relationship with the developer knowledge and experience, and the developer experience is likely to be related with the code quality produced.

Of course, best practices are not only related to naming. They can also be patterns for solving certain problems, working methodologies, etc. In those cases, they might be directly related to other quality attributes like performance and so on.

In the end, it seems plausible to believe that there is a correlation between these two variables: the quantity of best practices followed and the overall quality of the project.

Later in this document, this relation between these variables relation is proved.

5.3 Best Practices Examples

In this section, different best practices categories are identified and some examples listed. These are best practices for the Ruby community, but some of them can be applied to other communities. Best practices can be related to different things.

Best practices related to code formatting:

- *Use two spaces to indent code and no tabs*. Every worthy Ruby developer does it that way.
- *Remove trailing whitespace*. Trailing whitespace creates noises in version control systems.

Related to syntax:

- *Avoid return where not required*.
- *Suppress superfluous parentheses* when calling methods, but keep them when calling ”functions” (when you use the return value in the same line).

Related to naming:

- *Use snake_case for methods*.
- *Other method naming conventions*: Use map over collect, find over detect, find_all over select, size over length.

Specific to a framework (Ruby on Rails in for the following examples):

- *Law of Demeter*. A model should only talk to its immediate association.
- *Move code into controller*. According to MVC architecture, there should not be logic codes in view.
- *Isolate seed data*. Do not insert seed data during migrations. A rake task[1] can be used instead.
- *Do not use default route*, when using a RESTful design. The default RoR routes can cause security problems.
- *Replace Complex Creation with Factory Method*. Sometimes you will build a complex model with params, current_user and other logics in controller, but it makes your controller too big. You should move them into model with a factory method.

[1] Rakefiles work in similar way to Makefiles but are written in ruby. It is a simple way to write code to automate repetitive tasks.

5.4 Ruby on Rails Best Practices

Ruby and Ruby on Rails community members are, in general, known as being addicted to best practices. In reality, many of those best practices are studied development methodologies. As previously said it is common to associate Ruby on Rails with Behaviour Driven Development (BDD) and Agile methodologies. In addition, most Ruby developers, even language newcomers, are worried about following Rails philosophies and best practices. The majority of Ruby on Rails book authors speak about convention over configuration. Those conventions can be be seen as best practices too.

Because of all this, the Rails community has great potential to be a starting point to understand the role of best practices, what benefits come from using it, and how to measure best practices.

To start our studies, the first step is to define the community: Rails developers. After that, we need to find procedures or methodologies that most people in the community agree to consider it as best practice. The simplest way to achieve this seems to be by asking people in the community. In fact, there was already some work done in this manner. The web site Rails Best Practices[2], works in a similar way to a web forum and its objective is to engage developers to discuss which practices should be considered best practices to follow when building a Rails web application. At Rails Best Practices Web Site, every Rails developer can suggest new best practices, improve and comment suggested best practices, and vote whether he considers it a best practice or not. In Figure 5.1, one can see a sample page from this web site. The purpose of this page is to enable discussion about a particular best practice. One can quickly determine the current number of positive votes (23 in this case), number of comments, people that marked it as their favorite, and so on.

After deciding that some *procedure* is a *best practice*, it would be handy to find a way to automatically verify whether that practice is being followed by the developers of a given project. With that in mind, an open source ruby gem[3], called rails_best_practices, was created (by the authors of Rails best practices web site) based on the best practices with the most votes. After installing the gem, one can run it on any Rails project and it will automatically produce a report that shows where in the source code a project is failing to obey to consensual best practices. It is important to notice that the rails_best_practices gem does not indicate if a project is following best practices; it does the opposite and shows where the project is failing.

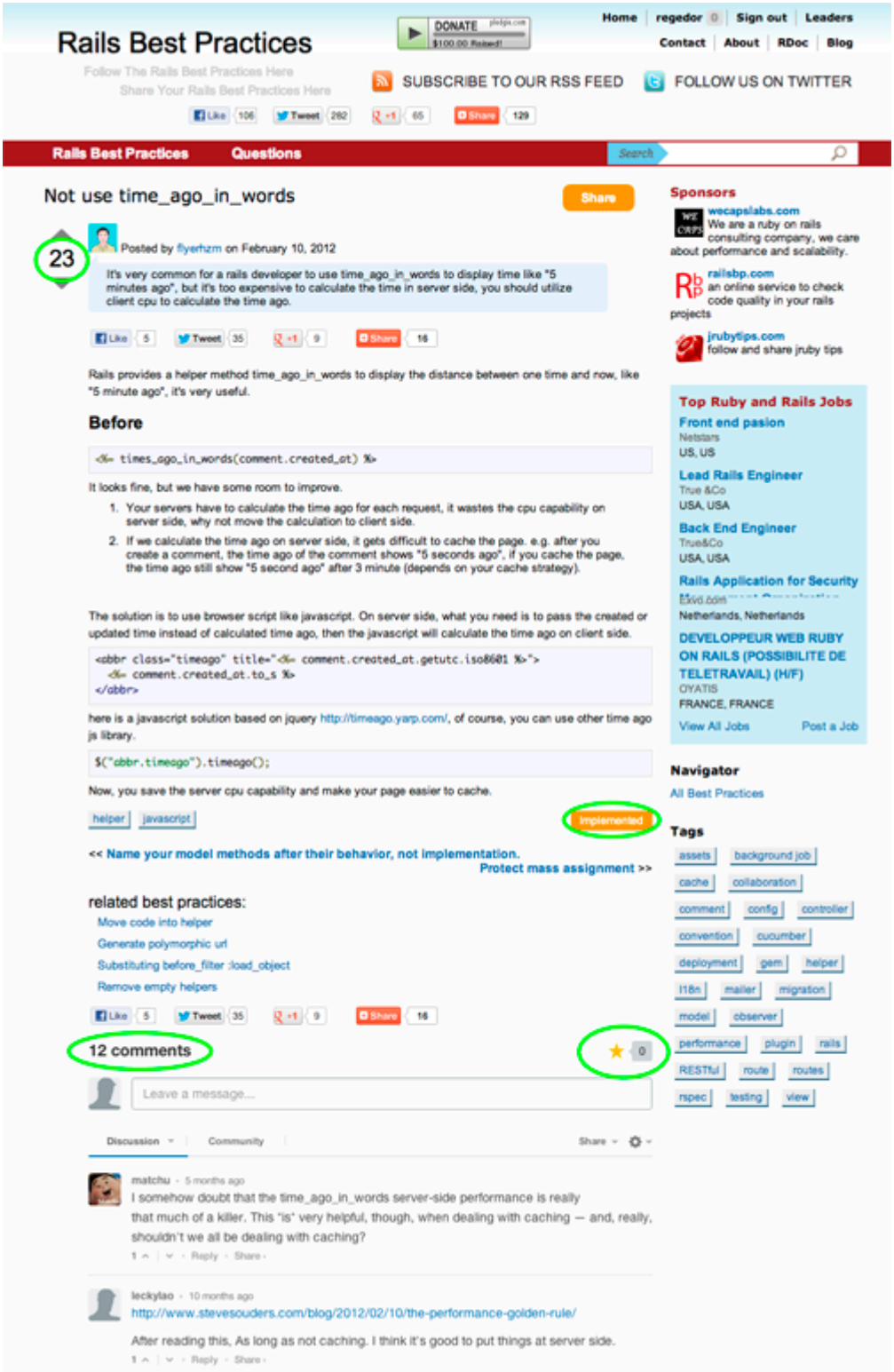
At the moment of writing, this gem can check for 33 different best practices from more than 70 described in the web site. One can read more details about these 33 best practices in Appendix A.

During the studies carried out during this thesis, we end up forking this project as a starting point for the creation of global best practices metric. We also did a few commits to this gem source code, now in the official version, to improve it and correct some bugs.

[2] http://www.rails-bestpractices.com/ is a web site created by Richard Huang, it was inspired by Wen-Tien Chang talk given at Kungfu RailsConf 2009 in Shanghai. Slides can be found here http://www.slideshare.net/ihower/rails-best-practices.

[3] Ruby Libraries are called gems. Ruby gems can be easily managed using rubygems (rubygems is for Ruby as aptitude is for Debian or cpan for perl).

Figure 5.1: Rails Best Practices (best practice discussion page)



All the source code is open source and can be found at GitHub (http://www.github.net/regedor). Everybody is welcome to contribute to this project by writing more best practices failure detectors, forking, correcting bugs, or improving it in any way.

Chapter 6

Assessing Ruby on Rails Projects

To assess the quality of a software project is not an easy task. Too many variables should be taken into consideration and most of their values can be considered subjective.

Our thesis is that the quality of an open source project is somehow influenced by the best practices followed by its contributors.

Furthermore, since the Rails community has been our object of study, we have decided to conduct a series of studies with the objective of finding correlations between best practices defined by the Rails community and the quality of projects following it.

With the objective of being able to automatically verify whether or not best practices are being followed by a given Rails project, the open source ruby gem rails_best_practices was created (by the authors of Rails best practices web site). We agreed on using it as the starting point for our work.

This chapter reports the different studies carried out, the main difficulties, and the results obtained.

6.1 First Study

One of the first things that we have noticed, when we applied the rails_best_practices gem to OSS projects, is that the biggest and most renowned projects have much more errors than the smaller and unknown projects. This apparent nonsense has a simple interpretation. Small projects (like the majority of Rails projects found in GitHub) are simple software packages, often developed by a single user, and are carried out for simple learning purposes. These applications are so simple that many times the code is almost entirely created by Rails code generators. Usually, when code is not written by humans, it has few mistakes concerning those recommendations.

Having taken the above into account, we decided to run the rails best practices gem on similar Rails projects. Seven *time tracking* or *project management* open source systems were chosen. After running the gem and counting the not best practices (NBPs)[1] occurrences, the following

[1] In fact, Rails Best Practices gem does not find best practices in the source code. It does the opposite, it discovers when the code is not written according to a best practice, in other words, it identifies bad practices (similar to the detection of code smells). We decided to name those occurrences NBP.

results were obtained:

Rails Best Practices Results							
Best Practice	**A**	**B**	**C**	**D**	**F**	**G**	**H**
Add model virtual attribute	-	2	7	-	-	5	4
Always add db index	-	-	-	43	-	-	51
Isolate seed data	-	-	-	-	-	79	17
Law of demeter	20	38	45	6	30	164	85
Move code into controller	-	-	-	-	2	-	4
Move code into model	-	26	-	7	1	3	19
Move model logic into model	-	-	76	11	11	98	100
Move finder to named_scope	-	4	9	2	4	25	-
Needless deep nesting	-	-	-	1	-	-	-
Not use default root	-	1	1	-	1	1	1
Notes use query attribute	-	2	-	-	-	-	-
Overuse route customizations	-	-	2	4	-	2	2
Remove trailing whitespace	68	57	126	110	330	316	100
Use factory method	-	15	9	5	1	8	19
Replace instance var with local var	13	-	70	239	142	31	100
Use before_filter	-	7	9	8	8	19	23
Wrong email content_type	-	3	-	-	-	-	-
Use query attribute	-	-	11	5	8	29	6
Use say with time in migrations	-	-	24	-	10	23	56
Use scopes access	-	-	-	-	-	-	04
User model association	-	-	12	9	-	1	21
Keep finders on their own model	8	4	1	-	11	-	-
Total	109	156	402	450	559	834	864

Results obtained by running the *best practices analyzer gem* on the 7 Open Source Projects chosen (data produced on April, 2011).
A: Rubytime , *B:* Notes , *C:* Tracks , *D:* Handy Ant , *F:* Retrospectiva , *G:* Redmine , *H:* Clockingit
Figures shown represent the number of times a project do not follow a best practice; is expected that *smaller the number, better the project*.

Table 6.1: Best practices analyzer gem raw results using 7 open source projects

Rubytime seems to have the best results and Clockingit the worst. The fact is that very good user reviews can be found about Rubytime. However, Tracks obtained an unexpected high score, since it has been very sparsely maintained (old code has higher probability of not following the current best practices). As explained before, those values are not really measuring if a project follows best practices , but instead measuring when it fails. This should also be taken into consideration.

The most evident problem here is that best practices are not being weighted, and the size of the project is not being considered. For instance, if the developers have the habit of leaving trailing white spaces, this will obviously be related to the size of the project. On the other hand, it is a best practice to remove the default route generated by rails; independent of the project size this is either true or false: there is no way to leave the route two times. So, if developers do not take into account the differences between these two best practices, when the project grows, the number of trailing spaces will increase and the results will show more NBPs. However, the other best practice (remove the default route) will always count as only one NBP. This can generate twisted results.

To avoid this, the projects were sized. The size attribute is based on the quantity of models and controllers in the project. After that, we divided the values previously obtained by the project size.

Now a new set of results emerge.

Rails Best Practices Results							
Best Practice	**A**	**B**	**C**	**D**	**F**	**G**	**H**
Total	109	156	402	450	559	834	864
Total Without Trailing Whitespace	41	99	276	340	229	518	764
Project Size	12	11	11	29	26	58	31
Total / Project Size	9	14	37	16	23	15	28
Total Without Trailing Whitespace / Project Size	3	9	25	12	9	9	25

A: Rubytime ; *B:* Notes ; *C:* Tracks ; *D:* Handy Ant ; *F:* Retrospectiva ; *G:* Redmine ; *H:* Clockingit
Results obtained by running the *best practices analyzer gem* on the 7 Open Source Projects chosen, after normalization (data produced on April, 2011).

Table 6.2: Best practices analyzer gem normalized results using 7 open source projects

Those results (in Table 6.2) are much more likely to be helpful in terms of understanding whether or not a project is following best practices. The numbers reflect both the community reviews and our own estimates much more.

6.2 Second Study

After the first study reported above, we felt that it was time to conduct a larger one. We felt that we should repeat the experiment over a larger sample. As a second target for this new phase, it was decided to find an objective quality rate (a reputation ranking) for each project in the sample. There was the need to define an objective quality metric to compare the metrics results with. This way it would be possible to prove that there is a statistical relationship between the quality of the project and the results of our best practices metric based on NBPs.

For the second study, we selected 40 Ruby on Rails projects hosted in GitHub and decided to consider the number of followers[2] and forks[3], that each project has on GitHub, as a *project reputation* metric.

The objective was then to prove that a negative correlation exists between the NBPs of a project and its followers and forks.

The previous study has shown us the need to apply different weights to each NBP. By dividing the NBPs by the size of project, we achieved better results. However, not all NBPs depend on the project size. It turned out to be clear, that each NBP should be weighted in a different way depending on the nature of the best practice related to it, and characteristics of the project. Therefore, we altered the rails best practices gem to make it possible to know how many project files were analyzed by each rails best practice checker. What this means is that if an NBP checker is just trying to find occurrences of errors in the models files of the project we will weigh this result

[2] Number of users that want to receive notifications about the project.
[3] Number of people that forked the project. This means that either they want to contribute to the project or create a derived project

based on the number and size of models. This is not the perfect solution yet, but it gives much better results than dividing all the best practices by the project size.

In the first study, we analyzed 7 projects. Now we have 40, and obviously this time we had to start automating some things. The work flow to get the information of a project is described in a few steps:

- *Retrieve GitHub information*. In this step we get the followers and forks(and more info that might be used in further analyses).
- *Download the project repository*.
- *Run rails best practices gems*. At this point, we get the non weighted NBPs and files given by each one of the 29 checkers.
- *Calculate the Weighted Global NBPs*. The evaluation algorithm consists of dividing the value returned by each NBP checker by the number of files checked, and then summing it.

After collecting the GitHub URLs for the 40 projects, we used a script to apply the described steps to each project and stored all the information about the projects in a CSV table.

Rails Best Practices Results											
Projects	**Forks**	**Watchers**	C1	C1 F.	**W. C1**	C2	C2 F.	**W. C1**	...	T. NBPs	**W. T. NBPs**
Rails Admin	30	2478	0	141	**0**	0	37	**0**	...	50	**739**
Rubytime	12	82	24	161	**149**	0	134	**0**	...	146	**1334**
Redmine	30	1781	49	996	**49**	1	362	**2**	...	884	**1402**
BrowserCMS	30	784	11	234	**47**	0	216	**0**	...	268	**1510**
Tracks	17	87	46	842	**54**	15	271	**55**	...	569	**2810**
...	...	...	...	...	...	...	...	...	...	...	...

Results obtained by running the *best practices analyzer gem* on the 40 Open Source Projects chosen, from GitHub (data produced on April, 2011). The full table can be found at www.study.gorgeouscode.com
C(x): The rails best practices gem has 29 checkers(when this study was carried), each one tries to find occurrences of a different nbp in the project.
C(x) Files: The number of files in the project, where it tried to find nbps (for instance, some checkers may only be concerned with html files, some other checker nbps my only occur in model files, etc)
W. C(x): Weighted C(x) = C(X) / C(x)Files * 1000 (A really small number is added to each variable to avoid divisions by zero).

Table 6.3: Best practices analyzer gem results using 40 open source projects

Table 6.3 is an excerpt of the obtained table. The full table can be found online:
http://study.gorgeouscode.com/files/rbp-study.pdf

6.3 Results

After building the table containing the results for the 40 projects, we began searching for a correlation between the different values. It was easy to find that our best practice metric was strongly related with the number of forks and watchers of the project. We discovered that the average

correlation index for the weighted C(x) columns is -0.2. Only three of the weighted C(x) columns do not have negative correlation, which is a pretty good result. Moreover, the positive correlation shown in these three columns is due to the fact that their respective checkers (without negative correlation) are aimed at finding NBPs that almost none of the projects were committing. This explains why there is no correlation. It is obvious that if every project follows a best practice, we can not really use it to distinguish the quality of the project; nevertheless, it makes it even more clear that it is a project that should be followed. In the end, when you do not consider those best practices, this relation is even more substantial.

The most important results are in the next table:

Correlations		
	Total NBPs	**Total Weighted NBPs**
Forks	0.14	-0.53
Watchers	0.07	-0.40

www.study.gorgeouscode.com for the complete table.

Table 6.4: Relations between NPBs forks and watchers

These correlation indexes show that if we just count the NBPs, there is no relation between them and the number of forks and watchers. Nevertheless, the Weighted NBPs have quite the perceptible negative correlation, both with watchers and forks.

By observing Table 6.4, it is possible to notice that the forks correlation is bigger. We believe this happens because forking a project shows intentions of digging into the code and, it is obviously easier to understand code from other people when it follows best practices.

After proving that this correlation truly exists, we now feel confident that the Weighted NBPs can be used as a metric of quality for Rails projects. In fact, we have achieved a new metric for classifying the quality of Rails projects in terms of maintainability, Not only this, as it was previously said in this document, but many of the best practices proposed by the Rails Best Practices Project members are also related with performance improvements.

However, there is one problem when using the Weighted Global NBPs to certify the quality of a Rails project. Although it is intuitive to understand that a small number in this metric is a good result (meaning that the project failed few times in terms of best practices), sadly no project of normal size was found with zero occurrences of NPBs. Consequently, simply saying that a project has 450 Weighted Global NBPs makes it harder for a user or developer to understand if this is a good or bad result.

Zero is definitely the best possible score, but there is no limit for the worst possible value. To retrieve some real meaning from this metric, we need to compare the project results with other ones. Therefore, a simple solution to this would be to analyze as many rails projects as possible and to determine and average value. After that, we would have a reference value from which to consider a single project above or under the average.

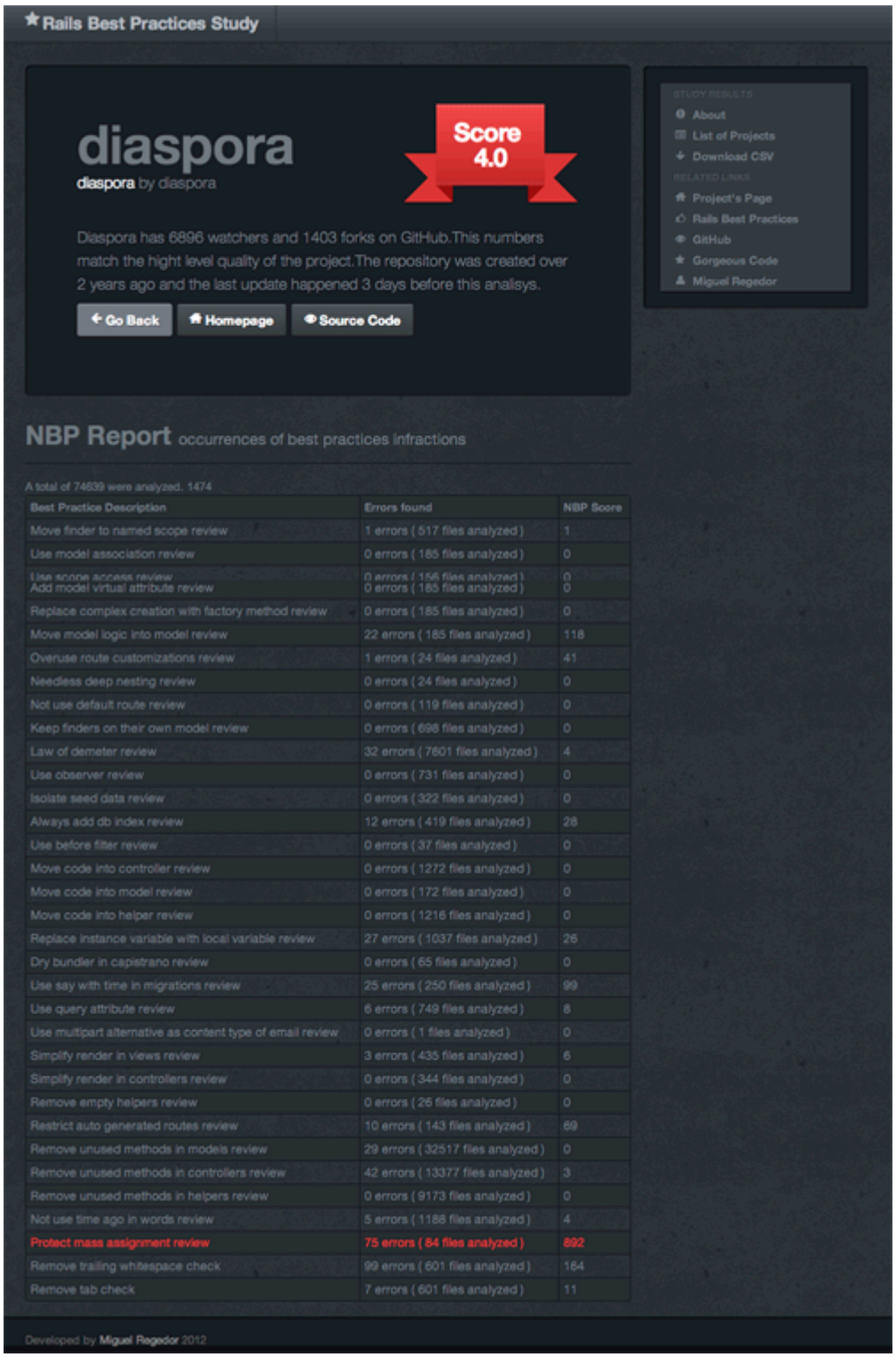
That is exactly what we did next: we created a simple web application capable of certifying the

quality of Rails projects based on the best practices followed. By using this database of reports, it is possible to dynamically update the average value for the quality of a project and use it as a reference value. It is also essential to eliminate any outliers during this process. Putting it all together is just a matter of inverting the Weighted Global NBPs of projects and scaling them. In the end, we can give a final score to any Rails project in a scale from 1 to 5, 5 being the best possible score.

The mentioned application was developed using Ruby on Rails, and it is actually possible to consult an extensive list of analyzed projects online at `app.study.gorgeouscode.com`. For each project, a page like the table in Figure 6.1 is generated, which contains the overall score, a brief possible explanation for the actual score, and a table pointing out the NBPs found. If a row is shown in red, it indicates that the score of the project was significantly lowered because it does not follow that specific best practice. In other words, this should be the first place to improve the code, which will also increase the quality of the project.

This application was the ultimate result of these studies.

Figure 6.1: Ruby on Rails Best Practices Report



Chapter 7

Conclusion

As it was said in the beginning of this document, thousands of open source software packages can be found online and freely downloaded in platforms like GitHub, the web-based hosting service for projects using the Git revision control system that hosts more than 1 million open source projects.

Throughout this study the need for assessing the quality of these open source software projects became clear. Maintainability turned out to be a crucial attribute in assessing their quality, obviously because of the community-driven orientation of these types of projects, described in chapter 2. However, this statement does not mean that maintainability is not also a quality attribute of great importance for closed source software projects as well. In fact, most standards, studies and tools within the field of measuring the quality of software projects were conducted by private companies with the purpose of improving and assuring the quality of their projects, which were usually closed source software projects. This idea together with the concepts of software quality and quality attributes were explored in chapter 4.

Few people have the ability to quickly assess the quality of a project by looking at source code. An application, able to perform automatic analysis of software projects and to generate a high-level overview of the code, is beneficial for assessing the quality of general software projects, independent of whether the license is open source.

Nevertheless, open source and closed source usually end up having a different development processes, and because of this they should obviously be analyzed in different ways. In the open source world there are no strict development rules; however, as time goes by and projects get bigger, something similar to rules or standard behaviors start to emerge in an spontaneous way, called best practices.This topic was discussed in chapter 5. There is the belief that by following these best practices, better results will be achieved.

In fact, the existence of common guidelines in software projects intuitively seems to be a potenciator of their maintainability to every developer. But just like popular sayings, their value is ambiguous if there is no scientific proof and its application could be considered a matter of taste.

However, we strongly believe that best pratices hold an extremely important value. Best practices are not rigid as standards, which means that their usage is also more flexible. Despite this, our belief since the beginning of this investigation is that if a project follows best practices, it has a higher probability of being a better project when it comes to quality standards. Following this idea

we focused our research towards proving the existence of a correlation between the quality of the GitHub open source projects and the amount of best practices followed, as explained in detail in chapter 6.

Initially, when we first started to elaborate this work it seemed like there was no work done concerning the measurement of best practices by automatic source code analysis. As a good surprise and back-up for the ideas described in this document, simultaneous with this research, several projects appeared on the scene based on similar ideas and concepts.

In the particular context of Ruby Community, we found that some efforts have already been made in this exact direction, defining best practices and creating scripts to automatically analyze the source code. A project called Rails Best Practices, started by Richard Huang, managed to put into practice some of our initial ideas of defining best practices with the help of the Rails developers' input and opinions, and even started to create code analyzers to detect whether a project is following them. The aforementioned project is described in chapter 5.

However, like most of the reports generated by the existent source code analyzers, the Rails best practices gem implementation is only spotting the occurrence of bad smells: not really identifying the best practices, but instead those that are not best practices. In chapter 6 we named this concept concept of NBPs. This way it is possible to verify whether a best practice is not being followed, since most of the time it is easier to identify incorrection instead of correction. This turns out to be a clever idea. These NBP reports are helpful, although not enough, but is a fact that allows the improvement of rails projects. There existed the need to interpret those results to end up with a high level quality statement. It was expected that by analyzing a massive amount of open source projects, it would be possible to start understanding the meaning of the values given by simple metrics, and thus to judge the impact of these numbers in the quality of the project. Consequently, we would find new ways to assess the quality, which is closely related to the level of maintainability, of an open source project. The final objective was to create a new higher-level metric capable of quantifying the best practices followed by a given project; something as simple as ”from 1 to 5, this project has 4 stars in terms of best practices”.

To achieve this we started looking for Rails projects hosted in GitHub, applying the different metrics and running the NBP checkers, and gathering all this information. As it was said before, by itself those values are useless; yet by comparing projects, it was possible to start understanding how to interpret those values. For instance, obvious things like considering the *size of the project* (it is intuitive that a project with 10 lines of code and 10 errors is worse than a project with 1000 and 20 errors), made a huge difference when taken into consideration.

From the studies we carried out, we also have learned that each best practice has a different importance level: 1 NBP that affects security or performance is, for sure, worse than 10 NBPs related to indentation; or 10 NBPs related to naming conventions are worse than the indentation mistakes. However, one should not forget that as best practices defined by the community, its individual importance should also be defined by the community, which in this case includes the

votes, comments, and reactions of users from the Rails best practices web site.

Lots of scripts were written to automate the code analysis process during the initial studies, and after analyzing more than 50 projects it was possible to find correlations between the usage of best practices described and discussed in Rails best projects, and the activity and popularity of projects in GitHub (number of project forks, project contributors and people following updates). Along with the paper, we gave arguments in order to emphasize that it is worthwhile to detect on the source code whether the author follows the best practices recommended by the respective community. After finding correlations, it was undeniable evidence of this. Having proved this thesis it was straightforward to build a web application, that, given a set of projects, can generate an NBP report as well as a global score from 1 to 5 in terms of following best practices.

As future work, more correlations should be explored .Some of those variables have already been identified as of great value: the number of commiters, starting date of the project, last commit data, and total number of commits. Those variables are also strongly related with the forks, watchers, and in the end, the quality of the project. For instance, when the last commit of the project is too old, it may likely be the justification for a bad score. Since best practices are always being renewed, old projects can rarely follow it in an exemplary manner.

Some of these subtleties are already partly taken into account in the developed application, but for now, just as a way to justify the obtained score. There are countless variables that do require study and can possibly improve this project. Nevertheless, it was already possible to run the application against a big set of projects. More than 100 projects have already been analyzed. A list of analyzed projects can be found at `app.study.gorgeouscode.com`. The reports and the overall score can be of great help not only for users when comparing projects, but also for open source developers willing to contribute to those amazing projects.

Software projects, societies or any other point of human interaction do require some kind of order. Strict rules are accepted as necessary evil, the only solution seemingly always defined from the top of the pyramid, and always cutting the freedom. Without freedom, however, no advance is possible. Best practices are the minimal necessary order in a free world.

References

[Bev99] N. Bevan. Quality in use: meeting user needs for quality. *Journal of Systems and Software*, 49(1):89–96, 1999. Cited on page 26.

[BHK06] D. Binkley, M. Harman, and J. Krinke. Animated visualisation of static analysis: Characterising, explaining and exploiting the approximate nature of static analysis. In *6th International Workshop on Source Code Analysis and Manipulation (SCAM 06). Philadelphia, Pennsylvania, USA*, pages 43–52. Citeseer, 2006. Cited on page 11.

[Boe09] B.W. Boehm. *Software engineering economics*. IEEE, 2009. Cited on page 27.

[CAH03] K. Crowston, H. Annabi, and J. Howison. Defining open source software project success. In *Proceedings of the 24th international conference on information systems (icis 2003)*, pages 327–340. Citeseer, 2003. Cited on page 4.

[CdPL09] L. Chung and J. do Prado Leite. On non-functional requirements in software engineering. *Conceptual Modeling: Foundations and Applications*, pages 363–379, 2009. Cited on page 26.

[CDS86] S.D. Conte, H.E. Dunsmore, and YE Shen. *Software engineering metrics and models*. Benjamin-Cummings Publishing Co., Inc. Redwood City, CA, USA, 1986. Cited on page 27.

[CM05] S. Christley and G. Madey. Collection of activity data for sourceforge projects. *Notre Dame, IN, Dept. of Computer Science and Engineering, University of Notre Dame*, ?:?, 2005. Cited on page 11.

[CM07] A. Capiluppi and M. Michlmayr. From the Cathedral to the Bazaar: An Empirical Study of the Lifecycle of Volunteer Community Projects. *INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING -PUBLICATIONS- IFIP*, 234/2007:31–44, 2007. Cited on pages 4, 10, and 13.

[Coo86] Peter Cooper. GitHub officially launches: Git hosting A-Go-Go! http://www.rubyinside.com/github-officially-launches-git-hosting-a-go-go-853.html, 1986. Cited on page 11.

[Dro02] R.G. Dromey. A model for software product quality. *Software Engineering, IEEE Transactions on*, 21(2):146–162, 2002. Cited on page 31.

[FN99] N.E. Fenton and M. Neil. Software metrics: successes, failures and new directions. *Journal of Systems and Software*, 47(2-3):149–157, 1999. Cited on page 27.

[GKS+07] G. Gousios, V. Karakoidas, K. Stroggylos, P. Louridas, V. Vlachos, and D. Spinellis. Software quality assessment of open source software. In *Proceedings of the 11th Panhellenic Conference on Informatics*, Athens University of Economics and Business, Patission 76, Athens, Greece, 2007. Citeseer. Cited on pages 4 and 26.

[Hah02] R.W. Hahn. *Government policy toward open source software*. Brookings Institution Press, Washington, DC, USA, 2002. Cited on pages 3 and 9.

[HS02] T.J. Halloran and W.L. Scherlis. High quality and open source software practices. In *Meeting Challenges and Surviving Success: 2nd Workshop on Open Source Software Engineering*, 2002. Cited on pages 4 and 12.

[JJ86] C. Jones and C. Jones. *Programming productivity*. McGraw-Hill New York, 1986. Cited on page 27.

[Kan02] S.H. Kan. *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002. Cited on pages 25, 26, and 27.

[LK94] M. Lorenz and J. Kidd. *Object-oriented software metrics: a practical guide*. Prentice Hall, 1994. Cited on page 29.

[MA07] A. Marchenko and P. Abrahamsson. Predicting software defect density: a case study on automated static code analysis. *Agile Processes in Software Engineering and Extreme Programming*, 4536/2007:137–140, 2007. Cited on page 4.

[McC76] T.J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, pages 308–320, 1976. Cited on page 28.

[RE00] E.S. Raymond and T. Enterprises. The Cathedral and the Bazaar. *KNOWLEDGE, TECHNOLOGY AND POLICY*, 12:1–35, 2000. Cited on pages 3 and 10.

[WLCR07] Y. Wang, Q. Li, P. Chen, and C. Ren. Dynamic fan-in and fan-out metrics for program comprehension. *Journal of Shanghai University (English Edition)*, 11(5):474–479, 2007. Cited on page 28.

[YC79] E. Yourdon and L.L. Constantine. *Structured design. Fundamentals of a discipline of computer program and systems design*. Prentice Hall, 1979. Cited on page 28.

Appendix A

Rails Best Practices

In this appendix we list the best practices implemented by the rails best practices analyzer gem, when the studies described in this documents were carried out. All of this best practices are a subset from the best practices proposed by rails best practices website users.

A brief explanation of what the code analyzer does is given and the words of the person that first suggested it in the rails best practices web site is presented together with the number of comments and score obtained in the web site. The data presented below were collected during the year 2012.

A.1 Remove Tab

> *Using tabs can mess up the spacing since some IDE's use 4 spaces for a tab, while others use 2, and some people don't use tabs at all, a mix of tabs and spaces causes things to not line up in most cases.*
>
> *Richard Huang July 04, 2011*

It scored 1 point (user votes). The discussion page has 7585 views and 7 comments.
What does the code analyzer:
Makes sure there are no tabs in files.
The full discussion can be seen here:
http://rails-bestpractices.com/posts/81-remove-tab

A.2 Remove Trailing Whitespace

> *Trailing whitespace always makes noises in version control system, it is meaningless. We should remove trailing whitespace to avoid annoying other team members.*
>
> *Richard Huang December 02, 2010*

It scored 16 points (user votes). The discussion page has 15609 views and 11 comments.
What does the code analyzer:
Make sure there are no trailing whitespace in codes.

The full discussion can be seen here:
http://rails-bestpractices.com/posts/60-remove-trailing-whitespace

A.3 Add Model Virtual Attribute

Do not assign the model's attributes directly in controller. Add model virtual attribute to move the assignment to model.

Wen-Tien Chang July 21, 2010

It scored 5 points (user votes). The discussion page has 12866 views and 0 comments.
What does the code analyzer:
Make sure to add a model virual attribute to simplify model creation.
The full discussion can be seen here:
http://rails-bestpractices.com/posts/4-add-model-virtual-attribute

A.4 Always Add Db Index

Always add index for foreign key, columns that need to be sorted, lookup fields and columns that are used in a GROUP BY. This can improve the performance for sql query. If you're not sure which column need to index, I recommend to use http://github.com/eladmeidar/rails_indexes, which provide rake tasks to find missing indexes.

Wen-Tien Chang July 24, 2010

It scored 5 points (user votes). The discussion page has 15849 views and 16 comments.
What does the code analyzer:
Review db/schema.rb file to make sure every reference key has a database index.
The full discussion can be seen here:
http://rails-bestpractices.com/posts/21-always-add-db-index

A.5 Dry Bundler In Capistrano

There are a few posts told you how to integrate bundler into capistrano, but they are out of date now. After bundler 1.0 released, you can add only one line in capistrano to use bundler.

Richard Huang July 20, 2010

It scored 16 points (user votes). The discussion page has 20958 views and 4 comments.
What does the code analyzer:
Review config/deploy.rb file to make sure using the bundler's capistrano recipe.
The full discussion can be seen here:
http://rails-bestpractices.com/posts/51-dry-bundler-in-capistrano

A.6 Isolate Seed Data

Rails 2.3.4 provides db:seed task that is the best way to insert seed data for set up a new application.

Wen-Tien Chang July 24, 2010

It scored 11 points (user votes). The discussion page has 10030 views and 8 comments.
What does the code analyzer:
Make sure not to insert data in migration, move them to seed file.
The full discussion can be seen here:
http://rails-bestpractices.com/posts/20-isolating-seed-data.

A.7 Keep Finders On Their Own Model

Rails 2.3.4 provides db:seed task that is the best way to insert seed data for set up a new application.

Wen-Tien Chang July 23, 2010

It scored 5 points (user votes). The discussion page has 3038 views and 5 comments.
What does the code analyzer:
Review model files to ake sure finders are on their own model.
The full discussion can be seen here:
http://rails-bestpractices.com/posts/13-keep-finders-on-their-own-model.

A.8 Law Of Demeter

According to the law of demeter, a model should only talk to its immediate association, don't talk to the association's association and association's property, it is a case of loose coupling.

Wen-Tien Chang July 24, 2010

It scored 19 points (user votes). The discussion page has 10697 views and 13 comments.
What does the code analyzer:
Review to make sure not to avoid the law of demeter.
The full discussion can be seen here:
http://rails-bestpractices.com/posts/15-the-law-of-demeter.

A.9 Move Code Into Controller

According to MVC architecture, there should not be logic codes in view, in this practice, I will introduce you to move codes into controller.

Wen-Tien Chang July 24, 2010

It scored 10 points (user votes). The discussion page has 4558 views and 0 comments.
What does the code analyzer:
Review a view file to make sure there is no finder, finder should be moved to controller.
The full discussion can be seen here:
http://rails-bestpractices.com/posts/24-move-code-into-controller.

A.10 Move Code Into Helper

According to MVC architecture, there should not be logic codes in view, in this practice,I will introduce you to move codes into helper.

Wen-Tien Chang July 24, 2010

It scored 10 points (user votes). The discussion page has 5126 views and 2 comments.
What does the code analyzer:
Review a view file to make sure there is no complex options_for_select message call.
The full discussion can be seen here:
http://rails-bestpractices.com/posts/26-move-code-into-helper.

A.11 Move Code Into Model

According to MVC architecture, there should not be logic codes in view, in this practice, I will introduce you to move codes into model.

Wen-Tien Chang July 24, 2010

It scored 10 points (user votes). The discussion page has 5289 views and 8 comments.
What does the code analyzer:

Review a view file to make sure there is no complex logic call for model.
The full discussion can be seen here:
http://rails-bestpractices.com/posts/25-move-code-into-model.

A.12 Move Finder To Named Scope

Complex finders in controller make application hard to maintain. Move them into the model as named_scope can make the controller simple and the complex find logics are all in models.

Wen-Tien Chang July 24, 2010

It scored 11 points (user votes). The discussion page has 6591 views and 3 comments.
What does the code analyzer:
Review a controller file to make sure there are no complex finder.
The full discussion can be seen here:
http://rails-bestpractices.com/posts/1-move-finder-to-named_scope.

A.13 Move Model Logic Into Model

In MVC model, controller should be simple, the business logic is model's responsibility. So we should move logic from controller into the model.

Wen-Tien Chang July 21, 2010

It scored 3 points (user votes). The discussion page has 7078 views and 5 comments.
What does the code analyzer:
Review a controller file to make sure that complex model logic should not exist in controller, should be moved into a model.
The full discussion can be seen here:
http://rails-bestpractices.com/posts/7-move-model-logic-into-the-model.

A.14 Needless Deep Nesting

Some people will define 3 or more level nested routes, it's a kind of over design and not recommended.

Wen-Tien Chang July 22, 2010

It scored 4 points (user votes). The discussion page has 5947 views and 4 comments.
What does the code analyzer:

Review config/routes.rb file to make sure not to use too deep nesting routes.
The full discussion can be seen here:
http://rails-bestpractices.com/posts/11-needless-deep-nesting.

A.15 Not Use Default Route

If you use RESTful design, you should NOT use default route. It will cause a security problem. I explain at http://ihower.tw/blog/archives/3265 too.

Wen-Tien Chang July 22, 2010

It scored 9 points (user votes). The discussion page has 4215 views and 0 comments.
What does the code analyzer:
Review config/routes file to make sure not use default route that rails generated.
The full discussion can be seen here:
http://rails-bestpractices.com/posts/12-not-use-default-route-if-you-use-restful-desi

A.16 Not Use Time Ago In Words

It's very common for a rails developer to use time_ago_in_words to display time like "5 minutes ago", but it's too expensive to calculate the time in server side, you should utilize client cpu to calculate the time ago.

Richard Huang 10 February, 2012

It scored 23 points (user votes). The discussion page has 13550 views and 11 comments.
What does the code analyzer:
Review view and helper files to make sure not use time_ago_in_words or distance_of_time_in_words_to_now.
The full discussion can be seen here:
http://rails-bestpractices.com/posts/105-not-use-time_ago_in_words.

A.17 Overuse Route Customizations

According to Roy Fieldingâ€™s doctoral thesis, we should use restful routes to represent the resource and its state. Use the default 9 actions without overusing route customizations.

Richard Huang 22 July, 2010

It scored 4 points (user votes). The discussion page has 3950 views and 0 comments.

What does the code analyzer: Review config/routes.rb file to make sure there are no overuse route customizations.

The full discussion can be seen here:
http://rails-bestpractices.com/posts/10-overuse-route-customizations.

A.18 Protect Mass Assignment

> *Rails mass assignment feature is really useful, but it may be a security issue, it allows an attacker to set any models' attributes you may not expect. To avoid this, we should add attr_accessbile or attr_protected to all models.*
>
> *Richard Huang 06 March, 2012*

It scored 7 points (user votes). The discussion page has 11969 views and 3 comments.

What does the code analyzer:

Review model files to make sure to use attr_accessible or attr_protected to protect mass assignment.

See the best practices details here:
http://rails-bestpractices.com/posts/148-protect-mass-assignment.

A.19 Remove Empty Helpers Review

> *If you use rails generator to create scaffolds or controllers, it will also create some helpers, most of the helpers are useless, just remove them.*
>
> *Richard Huang 09 April, 2011*

It scored 12 points (user votes). The discussion page has 8399 views and 6 comments.

What does the code analyzer:

Review a helper file to make sure it is not an empty module.

See the best practices details here:
http://rails-bestpractices.com/posts/72-remove-empty-helpers.

A.20 Remove Unused Methods In Controllers

Find out unused methods in controllers.

A.21 Remove Unused Methods In Helpers

Find out unused methods in helpers.

A.22 Remove Unused Methods In Models

Find out unused methods in models.

A.23 Replace Complex Creation With Factory Method

> *Sometimes you will build a complex model with params, current_user and other logics in controller, but it makes your controller too big, you should move them into model with a factory method.*
>
> *Wen-Tien Chang 21 July, 2010*

It scored 8 points (user votes). The discussion page has 4696 views and 0 comments.

What does the code analyzer:

Review a controller file to make sure that complex model creation should not exist in controller, should be replaced with factory method.

See the best practices details here:

http://rails-bestpractices.com/posts/6-replace-complex-creation-with-factory-method.

A.24 Replace Instance Variable With Local Variable

> *In partial view, we can use the instance variable directly, but it may be confused and make it hard to reuse anywhere, because we don't know exactly which instance variable can be used, so use the local variable in partial with explicitly assignment.*
>
> *Wen-Tien Chang 24 July, 2010*

It scored 23 points (user votes). The discussion page has 14592 views and 13 comments.

What does the code analyzer:

Review a partail view file to make sure there is no instance variable.

See the best practices details here:

http://rails-bestpractices.com/posts/27-replace-instance-variable-with-local-variable.

A.25 Restrict Auto Generated Routes

> *By default, Rails generates seven RESTful routes(new,edit,create,destroy,index,show, update) for a resource, sometime the resource only needs one or two routes, so just user :only or :except while defining routes to speedup the routing.*
>
> *Andy Wang 19 August, 2011*

It scored 12 points (user votes). The discussion page has 11901 views and 5 comments.

What does the code analyzer:

Review a route file to make sure all auto-generated routes have corresponding actions in controller.

See the best practices details here:

http://rails-bestpractices.com/posts/86-restrict-auto-generated-routes

A.26 Simplify Render In Controllers

Like the simplify render in views, from rails 2.3, we can also simplify render in controllers.

Richard Huang 12 December, 2010

It scored 8 points (user votes). The discussion page has 9773 views and 6 comments.

What does the code analyzer:

Review a controller file to make sure using simplified syntax for render.

See the best practices details here:

http://rails-bestpractices.com/posts/62-simplify-render-in-controllers.

A.27 Simplify Render In Views

render is one of the often used view helpers, we can pass object, collection or local variables. From rails 2.3, more simplified syntax for render are provided.

Richard Huang 04 December, 2010

It scored 8 points (user votes). The discussion page has 9773 views and 6 comments.

What does the code analyzer:

Review a view file to make sure using simplified syntax for render.

See the best practices details here:

http://rails-bestpractices.com/posts/61-simplify-render-in-views.

A.28 Use Before Filter

Don't repeat yourself in controller, use before_filter to avoid duplicated codes.

Wen-Tien Chang 24 July, 2010

It scored -6 points (user votes). The discussion page has 67294 views and 25 comments.

What does the code analyzer:

Review a controller file to make sure to use before_filter to remove duplicated first code line in different action.

See the best practices details here:
http://rails-bestpractices.com/posts/22-use-before_filter.

A.29 Use Model Association

Use model association to avoid assigning reference in controller.

Wen-Tien Chang 19 July, 2010

It scored 8 points (user votes). The discussion page has 10281 views and 7 comments.

What does the code analyzer:

Review a controller file to make sure to use model association instead of foreign key id assignment.

See the best practices details here:
http://rails-bestpractices.com/posts/2-use-model-association.

A.30 Use Multipart Alternative As Content Type Of Email

Rails uses plain/text as the default content_type for sending email, you should change it to multipart/alternative that email clients can display html formatted email if they support and display plain text email if they don't support html format.

Richard Huang 05 August, 2010

It scored 4 points (user votes). The discussion page has 10861 views and 3 comments.

What does the code analyzer:

Make sure to use multipart/alternative as content_type of email.

See the best practices details here:
http://rails-bestpractices.com/posts/41-use-multipart-alternative-as-content_type-of-email.

A.31 Use Observer

Observer serves as a connection point between models and some other subsystem whose functionality is used by some of other classes, such as email notification. It is loose coupling in contract with model callback.

Wen-Tien Chang 24 July, 2010

It scored 31 points (user votes). The discussion page has 15702 views and 7 comments.
What does the code analyzer:
Make sure to use observer (sorry we only check the mailer deliver now).
See the best practices details here:
http://rails-bestpractices.com/posts/19-use-observer.

A.32 Use Query Attribute

Do you always check if ActiveRecord's attributes exist or not by nil?, blank? or present? ? Don't do that again, rails provides a cleaner way by query attribute.

Richard Huang 03 October, 2010

It scored 19 points (user votes). The discussion page has 11265 views and 14 comments.
What does the code analyzer:
Make sure to use query attribute instead of nil?, blank? and present?.
See the best practices details here:
http://rails-bestpractices.com/posts/56-use-query-attribute.

A.33 Use say and say_with_time in migrations to make a useful migration log

Use say_with_time and say in migrations will produce a more readable output in migrations. And if use correctly it could be a helpful friend when something goes wrong because normally it is stored in the deploy log.

Gillermo 19 August, 2010

It scored 14 points (user votes). The discussion page has 5743 views and 2 comments.
What does the code analyzer:
Review a migration file to make sure to use say or say_with_time for customized data changes to produce a more readable output.
See the best practices details here:
http://rails-bestpractices.com/posts/46-use-say-and-say_with_time-in-migrations

A.34 Use Scope Access

You can use scope access to avoid checking the permission by comparing the owner of object with current_user in controller.

Wen-Tien Chang 20 JUly, 2010

It scored 16 points (user votes). The discussion page has 6775 views and 4 comments.

What does the code analyzer:

Review a controller to make sure to use scope access instead of manually checking current_user and redirect.

See the best practices details here:

http://rails-bestpractices.com/posts/3-use-scope-access.