Technical Debt Reduction Using a Game Theoretic Competitive Source Control Approach

Sarah Morrison-Smith
Dept. of Computer Science
Montana State University
Bozeman, MT, USA
sarah.morrisonsmith@msu.montana.edu

Stephen Dighans
Dept. of Computer Science
Montana State University
Bozeman, MT, USA
stephen.dighans@msu.montana.edu

Talon Daniels
Dept. of Computer Science
Montana University
Bozeman, MT, USA
talon.daniels@msu.montana.edu

Chad Marmon
Dept. of Computer Science
Montana State University
Bozeman, MT, USA
chad.marmon@msu.montana.edu

Clemente Izurieta
Dept. of Computer Science
Montana State University
Bozeman, MT, USA
clemente.izurieta@cs.montana.edu

Abstract

The management of technical debt and the use of productivity games are important aspects of developing software projects. A productivity game was created in the form of a competitive source control plug-in that rewards technical debt-reducing actions. The plug-in was tested by simulating source control usage with in a small sample project. Analysis showed that the plug-in appropriately assigned scores to developers for debt-reducing and debt-increasing actions. The plug-in has potential practical applications in the management of technical debt in workplace environments. The approach described in this paper is promising, and in future work we plan to test the *Build Game* plug-in with a wider variety of existing and simulated projects. Additional research is also planned to investigate the impact of the Build Game plug-in on workplace productivity.

Categories and Subject Descriptors
D.2.8 [**Software Engineering**]: Metrics

General Terms
Management, Measurement, Design, Experimentation.

Keywords
Source code control, software tools, technical debt, game theory, maintainability.

1. INTRODUCTION

Technical Debt (TD) is a metaphor created by Ward Cunningham to describe the burden placed on software engineers when shortcuts taken to speed development lead to long-term production setbacks [1]. Left unchecked, technical debt can impede agility [2], raise maintenance costs, and increase defects [3]. Thus, the management of technical debt is extremely important for any large software project.

Productivity games, *sensu* [4] (Director of Test at Microsoft), are a subcategory of games designed to improve the morale and productivity of employees. Microsoft's experiment, *42projects* [5], has definitively shown that games can increase productivity [6]. The use of productivity games is shown to enhance communication within the working environment and promote a high level of engagement [6]. Additionally, when used to enhance testing procedures, productivity games offer a means of regulating test coverage. Consequently, the use of productivity games is potentially very useful in any project, regardless of type.

Historically, source control systems have been enhanced to promote good programming practices by means of productivity game plug-ins. For example, the Continuous Integration Game plug-in by Rumfelt and Kutzinski [7] was created to decrease the number of times that a build becomes broken by awarding points to developers who commit builds with no failures. However, while the Continuous Integration Game incorporates additional test rules which focus on detecting suboptimal code with immediate consequences, no rules exist to penalize technical debt. Rumfelt and Kutzinski test and game rules for the Continuous Integration Build Game are described in tables I and II [7], [8], respectively.

Our objective is to improve upon this concept by creating a productivity game in the form of a competitive source control plug-in, called the *Build Game* plug-in, which rewards technical debt-reducing actions. This is accomplished by analyzing source control check-ins using predefined static analysis metrics. A contributor is assigned a score for every check-in. For example, decreasing complexity would be a positive action that yields a positive score, while reducing test coverage would be a negative action that yields a negative score.

TABLE I
THE CONTINUOUS INTEGRATION BUILD ADDITIONAL TEST RULES [7]

Action	Points
Adding/removing a high priority PMD Plug-in warning	−5/+5
Adding/removing a medium priority PMD Plug-in warning	−3/+3
Adding/removing a low priority PMD Plug-in warning	−1/+1
Adding/removing a high priority Task Scanner Plug-in task	−5/+5
Adding/removing a medium Task Scanner Plug-in priority task	−3/+3
Adding/removing a low priority Task Scanner Plug-in task	−1/+1
Adding/removing a Violations Plug-in violation	−1/+1
Adding/removing a Violations Plug-in duplication violation	−5/+5
Adding/removing a high priority Findbugs warning	−5/+5
Adding/removing a medium priority Findbugs warning.	−3/+3
Adding/removing a low priority Findbugs warning	−1/+1
Adding/removing a compiler warning	−1/+1
Adding/removing a Checkstyle warning	−1/+1

TABLE II
THE CONTINUOUS INTEGRATION BUILD GAME RULES [7] [8]

Action	Points
Breaking a build	−10*
Build with no failures	+1
Test failure (each)	−1
New passed test	+1

2. SOFTWARE CONFIGURATION

With every source code commit, the Build Game plug-in utilizes an external calculation of the amount of technical debt that a contributor creates or removes, and assigns a score accordingly. The plug-in uses five major components: Git [9] for source control, a Jenkins container [11], Apache Maven [12], a Sonar plug-in for Jenkins [13], and a Java Jenkins plug-in that assigns a technical debt score based on the Sonar analysis and keeps track of all the users' points.

2.1 GitHub and Git

GitHub is a web-based hosting service for software development using the Git revision control system [14]. This service provides source control management capabilities for the client [14]. In this project, GitHub is used to store revisions of committed builds.

2.2 Jenkins

Jenkins is an open source continuous integration tool written in Java [10]. Jenkins provides an easy-to-use system that makes it easier for developers to integrate changes into a project [11]. The plug-in for Jenkins requires the GitHub plug-in to be installed and using Git for source control.

2.3 Apache Maven

Apache Maven, a Java-based tool designed for building and managing Java-based projects, is an automation utility that allows developers to easily comprehend the state of a project [12]. The goal of Maven is to provide a unified build system that is easy and effective at keeping all users up to date of the build process [12]. Apache Maven plays a small role in our project. It builds the GitHub hosted project to be used by the Build Game plug-in and Sonar. Our plug-in for Jenkins is configured to only be compatible with Maven projects, thus it is a requirement that the user's project uses Maven.

2.4 Sonar

Sonar is an all-in-one, open platform designed to manage code quality [13]. Sonar is a web-based application that keeps a database of statistics derived from builds of a project that can be used in plug-ins that evaluate metrics. The Sonar plug-in for Jenkins is required for the purposes of this project. When setting up Sonar one must install the plug-in for technical debt. We use this plug-in to derive our metrics from the created technical debt database. Sonar's calculation of technical debt is configurable [15] depending on individual project needs.

2.5 Build Game Plug-in

The competitive source control plug-in Build Game, is the mechanism that converts raw Sonar analysis results into user scores and passes them to Jenkins to be recorded.

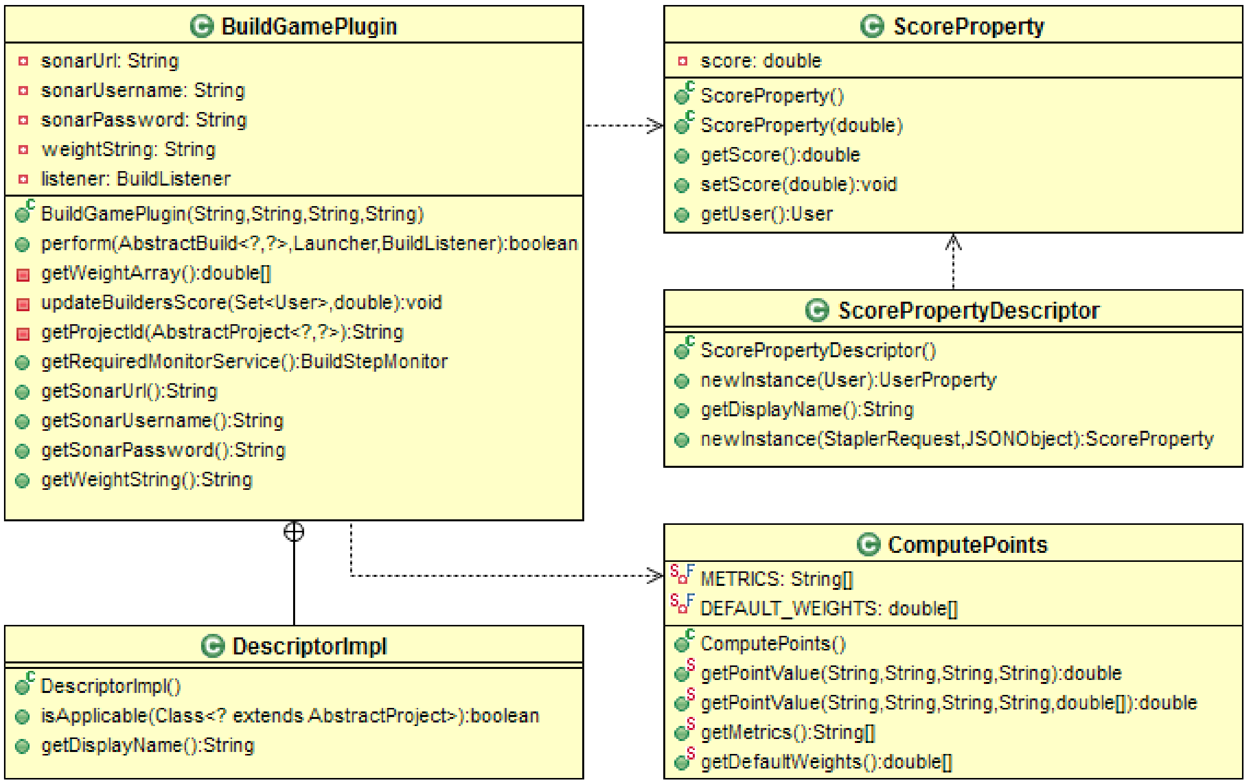
BuildGamePlugin
sonarUrl: String
sonarUsername: String
sonarPassword: String
weightString: String
listener: BuildListener
BuildGamePlugin(String,String,String,String)
perform(AbstractBuild<?,?>,Launcher,BuildListener):boolean
getWeightArray():double[]
updateBuildersScore(Set<User>,double):void
getProjectId(AbstractProject<?,?>):String
getRequiredMonitorService():BuildStepMonitor
getSonarUrl():String
getSonarUsername():String
getSonarPassword():String
getWeightString():String
ScoreProperty
score: double
ScoreProperty()
ScoreProperty(double)
getScore():double
setScore(double):void
getUser():User
ScorePropertyDescriptor
ScorePropertyDescriptor()
newInstance(User):UserProperty
getDisplayName():String
newInstance(StaplerRequest,JSONObject):ScoreProperty
ComputePoints
METRICS: String[]
DEFAULT_WEIGHTS: double[]
ComputePoints()
getPointValue(String,String,String,String):double
getPointValue(String,String,String,String,double[]):double
getMetrics():String[]
getDefaultWeights():double[]
DescriptorImpl
DescriptorImpl()
isApplicable(Class<? extends AbstractProject>):boolean
getDisplayName():String

Figure 1. UML class diagram of the Build Game Plugin. BuildGamePlugin is the main driver for the plugin; ComputePoints contains methods that calculate the point value to be awarded for the current build; ScoreProperty and ScorePropertyDescriptor, which are connected to and controlled by Jenkins, consist of methods that manipulate user scores; DescriptorImpl is a minor class nested in BuildGamePlugin that is used to implement a configuration/extensibility mechanism for the Jenkins user interface.

2.5.1 Files and Structure

The Build Game plug-in is comprised of five Java classes shown in Figure 1; a main driver; a class that consists of methods that calculate the point value to be awarded to the current build; two classes which are connected to and controlled by Jenkins that contain methods that hold, set, and retrieve a user's score; and a minor class that implements a configuration mechanism for the Jenkins user interface.

The Build Game plug-in also has several necessary peripheral jelly files. The jelly files are used by Jenkins to create configurable parameters for the plug-in [16]. These files set parameters for interfacing with Jenkins, and for setting configuration parameters. The Build Game plug-in also contains one important HTML file that defines the ordered list of sonar metrics used by the plug-in for the Jenkins web app.

2.5.2 Function and Behavior

The Build Game plug-in first performs a preliminary check to confirm that the outcome of the build is successful. Next, the project ID, Sonar URL, and weight string — a comma-delimited list that represents the importance and sign of each Sonar metric [17] — are retrieved from Jenkins. If an error occurs while retrieving the weight string, default weights are used. Since weighting is heavily dependent on individual projects and company needs, default weights assign the same value of importance to each metric. Default weights are given a positive or negative sign based on the color assigned to each metric in the Sonar dashboard. Positively signed weights are assigned to metrics colored green, while metrics that are colored red during negative variation are assigned negative weights. Next, the Build Game plug-in queries Sonar using the Sonar URL and retrieves the variation of measurements for each metric between the latest and previous build. Point values are calculated by the summation of each metric weight multiplied by its polled variation. Point values are then returned to the Build Game plug-in.

For example, if the weight string is (1.0, 1.0, −1.0, −1.0, 1.0) and the variation in Sonar metrics between two builds is (6.1, 17.0, −4.3, 5.2, −19.4), points are calculated by:

$$\begin{aligned}(1.0 * 6.1) + (1.0 * 17.0) + (-1.0 * -4.3) \\ + (-1.0 * 5.2) + (1.0 * -19.4) = 2.8\end{aligned} \tag{1}$$

During the final step, the set of contributors to the latest build are retrieved from Jenkins. The Build Game plug-in adds the current point value to each user's previous score.

3. METHODOLOGY

The Build Game plug-in was tested using a productivity game simulation over a small sample project of approximately 1500 lines of code. Simulation was used to test the correctness of the plug-in's logic prior to fully integrating with Jenkins. A Python script was created to simulate Jenkins by traversing the Git commit history, checking out a previous version, then running Sonar analysis. Each build was defined as a snapshot build. This allowed Sonar to store the data for every commit. After Sonar analysis was complete, the script then ran through the logical statements contained in the Build Game plug-in to provide the total score for the build.

4. RESULTS

The following figures describe trends in technical debt ratio and score over a period of 47 builds. Technical debt ratio is calculated by Sonar as:

$$(\textit{technical debt} / \textit{total possible debt}) * 100, \qquad (2)$$

where technical debt and total possible debt are both values calculated by Sonar [15].

In Figure 2 we display the overall technical debt ratio and scores over a period of 47 versions of the build.

Figures 3 to 7 describe the normalized influence of additional metrics on technical debt and score. These metrics are used by Sonar to calculate technical debt and are used by the Build Game plug-in to calculate the score. Metrics are subdivided into categories for readability.

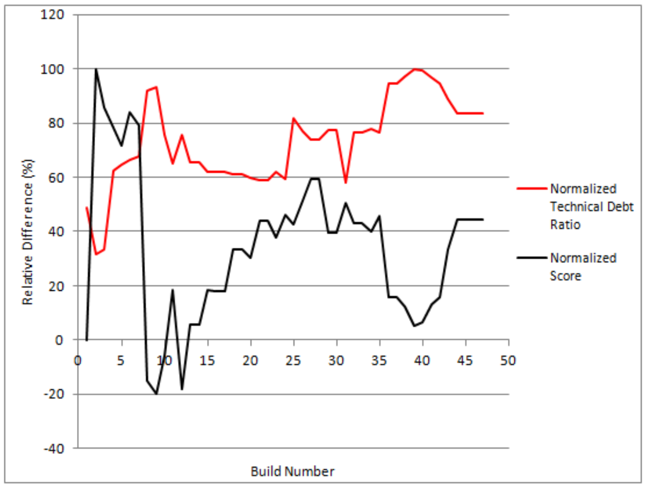

Figure 2. Normalized technical debt ratio and scores.

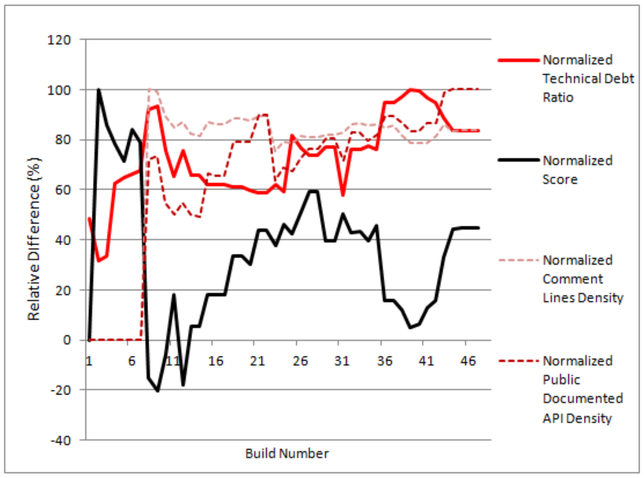

Figure 3. Normalized comment lines density, publicly documented API density, technical debt ratio, and scores.

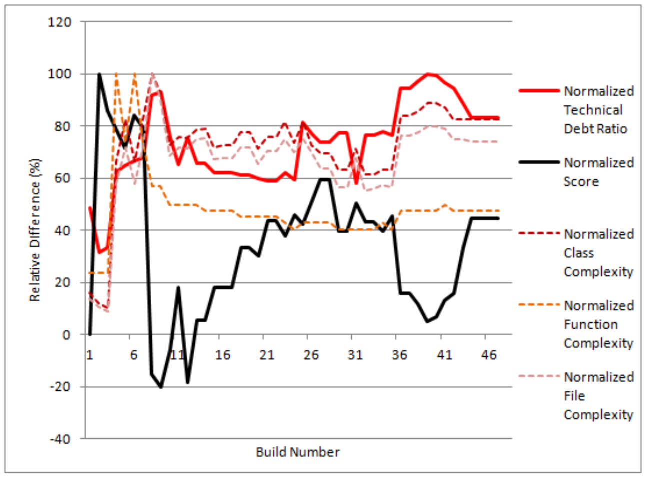

Figure 4. Normalized class complexity, file complexity, function complexity, technical debt ratio, and scores.

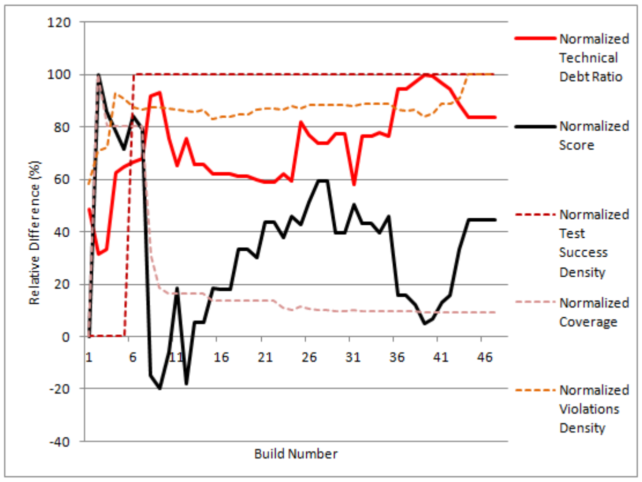

Figure 5. Normalized test success density, test coverage, violations density, technical debt ratio, and scores.

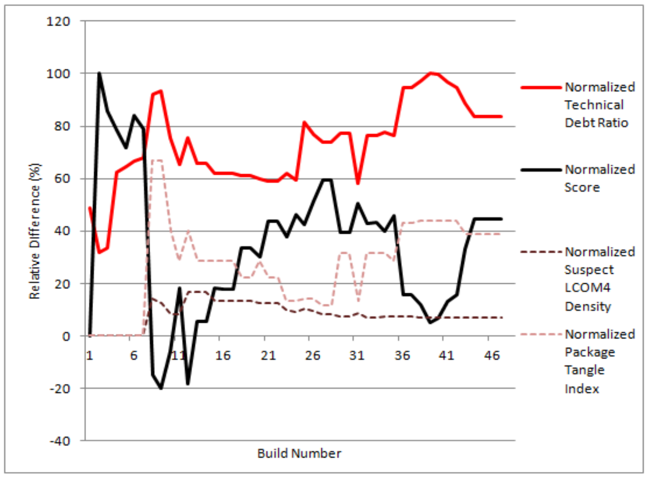

Figure 6. Normalized duplicated blocks, duplicated files, duplicated lines density, technical debt ratio, and scores.

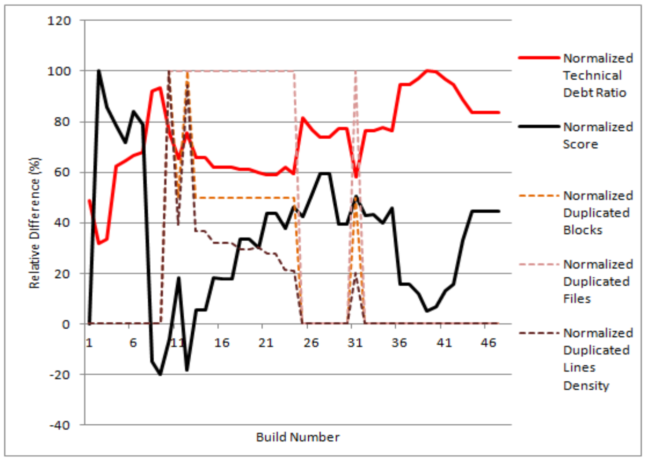

Figure 7. Normalized suspect lack of cohesion of methods (LCOM4) density, package tangle index, technical debt ratio, and scores.

5. DISCUSSION

As depicted in Figure 1, there is an inverse relationship between the score determined by the Build Game plug-in, and the technical debt ratio determined by Sonar. Although several weighted metrics are used to compute the score, which enhances the ability for users to customize, technical debt is appropriately represented by Sonar's technical debt ratio. The score/technical debt ratio suggests that the Build Game plug-in appropriately assigns points to developers based on their contributions to technical debt.

A spike in score and metrics is shown in figures 2 - 7 during builds 1 through 5. This is expected because going from no code to even the smallest code base reflects a large positive increase in points, boosting the developers' score. This is quickly equalized by build 7, as a gradual build-up of negative points counteracts the low rate at which positive changes occur. It is at this point that the build appears to stabilize. Since this spike can be expected at the beginning of every project, the Build Game plug-in can be employed at any time during the project and still be able to analyze past builds. This would initiate the game with a more accurate starting score.

With two notable exceptions, the relationship between individual metrics and score (indicated in figures 2 – 7) progressed as expected. Between builds 28 through 29 and 35 through 44, a large drop in score was not matched by an equivalent rise in technical debt. Since score is determined by the weighted contribution of several metrics in addition to technical debt, a sharp change in one heavily weighted metric can have a profound impact on the score. It appears, in these instances, that the change in score was caused by an increase in the *package_tangle_index* metric. This indicates that while the rest of the metrics used by the Build Game plug-in appropriately influence score, the *package_tangle_index* may be inappropriately weighted in the technical debt calculation. Because weights are intended to be tailored to the individual project or company, it is reasonable to allow a user to change the value of said parameters.

6. CONCLUSION

As the results have shown, the Build Game plug-in has the potential to be a reliable tool for deriving a consistent score, which leads to many practical applications. The game atmosphere fostered by the plug-in can help employers maintain an enjoyable work atmosphere, thus promoting positive employee morale. The plug-in's focus on decreasing technical debt facilitates decreased down time and promotes faster development cycles, thus improving employee productivity. In addition, the use of Open Source software used by Build Game makes it appropriate for decreasing technical debt in personal computing scenarios.

7. REFERENCES

[1] Cunningham, W. *The WyCash Portfolio Management System*. Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum), ser. OOPSLA '92. New York, NY, USA: ACM, 1992, pp. 29–30. [Online]. Available: http://doi.acm.org/10.1145/157709.157715

[2] Sussna, J. *Technical Debt, and Adaptive Organizations.* DevOps, 2012. http://blog.ingineering.it/post/21311393348/tech-debt

[3] Eisenberg, R. J. *A threshold based approach to technical debt.* ACM SIGSOFT Software Engineering Notes archive. March 2012, 37(2), pp 1. http://dl.acm.org/citation.cfm?id=2108151&dl=ACM&coll=DL&CFID=97542137&CFTOKEN=95173102.

[4] Smith, R. *Portfolio selection and game theory in defect prevention.*" July, 2009. http://blogs.msdn.com/b/microsoft_press/archive/2009/07/31/portfolio-selection-and-game-theory-in-defect-prevention.aspx

[5] 42Projects. *What is 42Projects?* http://www.42projects.org/

[6] Smith, R. *How Gaming Transforms the Workplace.* Casual Connect. Seattle, Washington, July, 2011. http://casualconnect.org/lectures/business/how-gaming-transforms-the-workplace-ross-smith/

[7] Ramfelt, E. and Kutzinski, C. *The Continuous Integration Game plug-in.* March 6, 2012. https://wiki.jenkins-ci.org/display/JENKINS/The+Continuous+Integration+Game+plugin

[8] Ramfelt, E. *The Continuous Integration Game plug-in*. March 15, 2011. http://wiki.hudson-ci.org/display/HUDSON/The+Continuous+Integration+Game+plugin

[9] Git. *Git*. September 26, 2012. http://git-scm.com/

[10] Magnyan and Kawaguchi, K. *Git Plug-in.* April 2012. https://wiki.jenkins-ci.org/display/JENKINS/Git+Plugin

[11] Kawaguchi, K. and Molter, T. *Meet Jenkins.* April 2012. https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins

[12] Apache Software Foundation. *What is Maven?* April 2012. http://maven.apache.org/what-is-maven.html

[13] SonarSource. http://www.sonarsource.org/

[14] GitHub Inc. *Github*. September 26, 2012. https://github.com/

[15] Gaudin, O. and Brandhof, S. *Technical Debt Calculation.* March 2011. http://docs.codehaus.org/display/SONAR/Technical+Debt+Calculation

[16] Oestreicher, S. *Basic guide to Jelly usage in Jenkins.* March 2011. https://wiki.jenkinsci.org/display/JENKINS/Basic+guide+to+Jelly +usage+in+Jenkins

[17] Mallet, F. and Madrikov, E. *Metric Definitions.* January 2012. http://docs.codehaus.org/display/SONAR/Metric+definitions