Towards an Open-Source Tool for Measuring and Visualizing the Interest of Technical Debt

Davide Falessi  
California Polytechnic State University  
San Luis Obispo, CA, USA  
dfalessi@calpoly.edu

Andreas Reichel  
Mannheim University of Applied Sciences  
Mannheim, Germany  
andreas.reichel@stud.hs-mannheim.de

Abstract — Current tools for managing technical debt are able to report the principal of the debt, i.e., the amount of effort required to fix all the quality rules violated in a project. However, they do not report the interest, i.e., the disadvantages the project had or will have due to quality rules violations. As a consequence, the user lacks support in understanding how much the principal should be reduced and why. We claim that information about the interest is, at least, as important as the information about the principal; the interest should be quantified and treated as a first-class entity like the principal. In this paper we aim to advance the state of the art of how the interest is measured and visualized. The goal of the paper is to describe MIND, an open-source tool which is, to the best of our knowledge, the first tool supporting the quantification and visualization of the interest. MIND, by analyzing historical data coming from Redmine and Git repositories, reports the interest incurring in a software project in terms of how many extra defects occurred, or will occur, due to quality rules violations. We evaluated MIND by using it to analyze a software project stored in a dataset of more than a million lines of code. Results suggest that MIND accurately measures the interest of technical debt.

Keywords — technical debt, quality rules, maintainability, defect proneness, interest.

I. INTRODUCTION

A. Context

Market opportunities bring economic benefits that help achieve project success. However, there are many examples of software development projects in which shortcomings made to meet market opportunities have resulted in large cost overruns, severe quality issues, inability to add new features without breaking existing features, and even the premature loss of a system. All these issues have a significant impact on profitability [1].

Technical Debt can be seen as the result of past software decisions that negatively affect the project future [1][2][3]. Steve McConnell defined technical debt as “A design or construction approach that's expedient in the short-term but that creates a technical context in which the same work will cost more to do later than it would cost to do now (including increased cost over time).” [4]

B. Terminology

Current tools that analyze and monitor technical debt use the concepts of quality rules and their violations. Specifically, a quality rule is an empirically validated software engineering principle that dictates how the code should be developed to support maintenance. Examples of quality rules include: avoiding code clones, high comments frequency, low code complexity, intuitive variable naming, etc. A quality rule violation is a piece of code that does not conform to a quality rule. Note that the same code can violate several rules, and the same rule can be violated in several pieces of code.

Technical debt consists of two main components: the principal and the interest. The principal is the cost required to fix violations. It is commonly measured or estimated as the effort for refactoring the code that violates the quality rules [5]. The interest is the cost of violating a rule, or of not fixing a violated rule. This is commonly measured or estimated as decreased productivity or extra defects [2].

Finally, refactoring means paying the principal or, in other words, improving software design to make it more understandable and changeable [6]. Therefore, refactoring coincides with removing violations.

C. Problem statement

We note that in an ideal world that has unlimited resources and knowledge of code development, projects would have the time to apply all quality rules or fix violations. In practice, knowledge, time, and effort are very limited and tradeoffs must be made between short-term benefits (e.g., customers satisfaction) and long-term ones (e.g., maintainability) [7][8]. For this and other reasons, technical debt accumulates over time in any software project [8].

Several tools have been developed to monitor technical debt. CAST¹ and SonarQube² are by far the two most used tools to monitor technical debt. Both tools are able to report the principal of the debt as the amount of effort required to fix all the rules

---

¹ http://www.castsoftware.com/products/application-intelligence-platform
² http://www.sonarqube.org/
violated in a project. However, the number of violations can reach hundreds or thousands; in this case, the principal can become extremely high and lose practical utility. For instance, current SonarQube analysis\(^3\) shows that the Apache project, consisting of around 4 million Lines of Code (LOC), has around 256 thousand violations, which result in a principal of around 7 thousand days.

CAST and SonarQube can also visualize the trend over time of the number of violations\(^3\). However, the consequences of violations are unclear. For example, we do not know the consequences of 5 Critical violations or even if these consequences are smaller or bigger than 10 Info violations.

In this paper we claim that information about the interest is at least as important as information about the principal. Hence, the interest should be quantified and treated as a first-class entity like the principal.

\section*{D. Aim}

In this paper we aim to advance the state of the art of how the interest is measured and visualized. The goal of the paper is to describe an open-source tool called MIND: Managing techNical Debt.

MIND is, to the best of our knowledge, the first tool supporting the quantification and visualization of the interest. Specifically, MIND, by analyzing historical data coming from Redmine and Git repositories, reports the interest incurring in a software project in terms of how many extra defects occurred, or will occur, due to violations.

\section*{E. Structure}

The reminder of the paper is structured as follows. Section 2 provides and overview of the methodology to measure the interest of technical debt whereas Section 3 describes how MIND can be used and interpreted. Section 4 validates the approach and Section 5 concludes the paper by reporting the major next steps and open issues.

\section*{II. METHODOLOGY}

MIND reports the interest in terms of several metrics. In this section we report the formal definitions and the methodology adopted to measure those.

\subsection*{A. Metrics}

\subsubsection*{1) Defect Proneness}

Equation 1 defines defect proneness (DP) as the number of defects injected divided by the LOC touched for a specific class \((j)\) in a specific release \((i)\). This variable can be seen as the probability of introducing a defect when modifying a single LOC, regardless of the type of modification (add, delete, or change). We believe that focusing on defects is a strong indicator of the pain felt due to technical debt; higher defect proneness usually implies extra effort (required to fix the extra defects) and decreased reputation of the company or product. We used the defect injection frequency, rather than simply the number of defects, because defects can be injected only if the class is touched. In this work we do not differentiate between types and severity of defects.

\[
DP_{ij} = \frac{\text{Number of defects}_{ij}}{\text{LOC touched}_{ij}}
\]

Equation 1: Definition of Defect Proneness (DP) for a specific class \((j)\) in a specific release \((i)\).

\subsubsection*{2) Maximum defects per 100 LOC touched}

Equation 2 defines the maximum defect per 100 LOC touched as one hundred multiplied by the highest extra defect proneness among the \(n\) classes of a specific release \((j)\).

\[
\text{MaxDP}_i = 100 \times \text{Max}_{j=1}^n (DP_{ij})
\]

Equation 2: Definition of Maximum Defects (MaxD) among classes \((j)\) of a specific release \((i)\) per each 100 LOC touched

\subsubsection*{3) Extra defect proneness}

Equation 3 defines extra defect proneness (EDP) as the defect proneness of a class \((j)\) minus the defect proneness of the same class without violations \((j^*)\) for a specific class \((j)\) in a specific release \((i)\). The EDP definition is consistent with the definition of interest provided by Vallery et al.\(^9\) because \(DP_{ij}\) represents the ideal situation in which a class has no violation.

\[
\text{EDP}_{ij} = DP_{ij} - DP_{ij^*}
\]

Equation 3: Definition of Extra Defect Proneness (EDP) for a specific class \((j)\) in a specific release \((i)\).

\subsubsection*{4) Maximum extra defects per 100 LOC touched}

Equation 4 defines the maximum extra defect per 100 LOC touched as one hundred multiplied by the highest extra defect proneness among the \(n\) classes of a specific release \((j)\).

\[
\text{MaxEDP}_i = 100 \times \text{Max}_{j=1}^n (EDP_{ij})
\]

Equation 4: Definition of Maximum Extra Defects (MaxED) among classes \((j)\) of a specific release \((i)\) per each 100 LOC touched

\subsubsection*{5) Relative extra defect proneness}

Equation 5 defines the relative extra defect proneness (REDP) as the defect proneness of a class \((j)\) minus the defect proneness of the same class without violations \((j^*)\) divided by the defect proneness of a class \((j)\) for a specific class \((j)\) in a specific release \((i)\).

\[
\text{REDP}_{ij} = \frac{DP_{ij} - DP_{ij^*}}{DP_{ij}}
\]

Equation 5: Definition of Relative Extra Defect Proneness (REDP) for a specific class \((j)\) in a specific release \((i)\).

\subsubsection*{6) Average relative extra defect proneness}

Equation 6 defines the average relative extra defect proneness (AREDP) as the average relative extra defect proneness among the \(n\) classes in a specific release \((i)\).

\[^3\] http://goo.gl/QyogK9
However, this is only 

\[ AREDP_i = \text{Average}^n_{j=1} \left( \frac{DP_{ij} - DP_{ij^*}}{DP_{ij}} \right) \]

Equation 6: Definition of average relative extra defect proneness (AREDP) in a specific release (i).

7) Violation density

Equation 7 defines violation density as the number of violations divided by size (in LOC) of a specific class (i) of a specific release (j).

\[ VD_{ij} = \frac{\text{NumberOfViolations}_{ij}}{\text{Size}_{ij}} \]

Equation 7: Definition of violation density (VD) for a specific class (j) in a specific release (i).

8) Linkage

Equation 8 defines linkage as the number of commits having a ticket number associated with it divided by the total number of commits.

\[ \text{Linkage} = \frac{\text{NumberOfCommitsWithTickets}}{\text{NumberOfCommits}} \]

Equation 8: Definition of the linkage in a dataset.

9) Estimation error

Equation 9 defines the estimation error as the absolute of current defect proneness minus the estimated defect proneness. This metric is generally called mean absolute error (MAE).

\[ MAE = \text{ABS(Predicted} - \text{Actual)} \]

Equation 9: Definition of Mean Absolute Error.

B. Measurement approach

MIND measures the interest in terms of how many extra defects occurred, or will occur, due to quality rules violations, i.e., extra defect proneness. According to Equation 3, this is defined as the defect proneness of a class (DP_{ij}) minus the defect proneness of the same class without violations (DP_{ij^*}).

When we analyze historical data, there are two cases: a class either had or did not have violations. In the latter case (no violations), the interest is zero by definition (i.e., \( DP_{ij} = DP_{ij^*} \)). In the case that the class had violations, we can directly measure its defect proneness (DP_{ij}); however, this is only one variable of Equation 3. We cannot measure the other variable, i.e., defect proneness of the class that did not have violations (DP_{ij^*}), because the class actually had violations. Therefore, the extra defect proneness of a class can be estimated rather than directly measured.

Equation 10 defines our model to predict the defect proneness of a class without violation, in case actually had violations. The specific prediction model used in MIND is Linear Regression which is a standard reliable model for estimating variables expressed in a ratio scale WEKA [10] [11].

\[ DP_{ij^*} = f(VD_{ij^*}) \]

Equation 10: Prediction formula to compute the defect proneness of a class without violation(i*) in a release (i).

According to Equation 10, in MIND the interest is assumed to vary according to the violation density. This assumption is validated in Section IV. Additional validation is provided in a recent case study showing that classes featuring a high violation density are on average five times more defect prone than classes with no violations [12]. The main advantage of using violation density, rather than number of violations, is to eliminate the effect of size, which is one of the most important influencing factors in maintainability [13] [14]. We also see violations as traps in the code. As such, the probability of having defects, i.e., falling into a trap, depends on the density of the traps rather than their number. We note that the use of violation density captures the concept of interaction effect among different quality rules, which turned out to be important during maintenance [15].

Figure 1 reports the approach for measuring the violation density and the defect proneness of a class in a specific release.

\[
\begin{array}{ccc}
\text{Violations}_i & \text{LOC touched}_i & \text{Defects}_i \\
\text{Size}_i & & \\
\hline
\end{array}
\]

\[
\begin{array}{ccc}
\text{Release i-1} & \text{Release i} & \text{Release i+1} \\
\hline
\end{array}
\]

Figure 1 Measurement approach, for dependent and independent variables, for a class in a release.

Specifically, the violations and size are computed at the start of a release by reading the SVN repository. LOC touched and defects are computed at the end of the release. Defects are measured by analyzing the commit comments in Git (to understand the specific class related to the defect) and the Redmine repository (to understand that the change is related to a defect). For instance, a commit in SVN can have a comment like “PROJ1-546”. MIND checks the type of the ticket in Redmine having the identifier “PROJ1-546” (i.e., “implementation” or “defect fix”) and then update the number of defects counter of the specific class affected by the commit.

Once measured, violation density and defect proneness are recorded in a tabular format similar to the one reported in Table 1. Afterward, this data is used for training and validating the prediction model.
Table 1: Sanitized example of the measurement used as input for the training and validation of the estimation model.

<table>
<thead>
<tr>
<th>Class ID</th>
<th>Version</th>
<th>Violations Density</th>
<th>Defect Proneness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class 1</td>
<td>v1.0</td>
<td>0.052</td>
<td>0.019</td>
</tr>
<tr>
<td>Class 2</td>
<td>v1.0</td>
<td>0.164</td>
<td>0.423</td>
</tr>
<tr>
<td>Class 3</td>
<td>v1.0</td>
<td>0.252</td>
<td>0.238</td>
</tr>
<tr>
<td>Class 4</td>
<td>v1.0</td>
<td>0.013</td>
<td>0.795</td>
</tr>
<tr>
<td>Class 5</td>
<td>v1.0</td>
<td>0.333</td>
<td>0.763</td>
</tr>
<tr>
<td>Class 6</td>
<td>v1.0</td>
<td>0.410</td>
<td>0.760</td>
</tr>
<tr>
<td>Class 1</td>
<td>v1.1</td>
<td>0.778</td>
<td>0.717</td>
</tr>
<tr>
<td>Class 2</td>
<td>v1.1</td>
<td>0.177</td>
<td>0.663</td>
</tr>
<tr>
<td>Class 3</td>
<td>v1.1</td>
<td>0.429</td>
<td>0.307</td>
</tr>
<tr>
<td>Class 4</td>
<td>v1.1</td>
<td>0.853</td>
<td>0.057</td>
</tr>
<tr>
<td>Class 5</td>
<td>v1.1</td>
<td>0.868</td>
<td>0.451</td>
</tr>
<tr>
<td>Class 6</td>
<td>v1.1</td>
<td>0.166</td>
<td>0.774</td>
</tr>
<tr>
<td>Class 1</td>
<td>v1.2</td>
<td>0.232</td>
<td>0.736</td>
</tr>
<tr>
<td>Class 2</td>
<td>v1.2</td>
<td>0.260</td>
<td>0.271</td>
</tr>
<tr>
<td>Class 3</td>
<td>v1.2</td>
<td>0.439</td>
<td>0.229</td>
</tr>
<tr>
<td>Class 4</td>
<td>v1.2</td>
<td>0.051</td>
<td>0.853</td>
</tr>
<tr>
<td>Class 5</td>
<td>v1.2</td>
<td>0.174</td>
<td>0.370</td>
</tr>
<tr>
<td>Class 6</td>
<td>v1.2</td>
<td>0.426</td>
<td>0.548</td>
</tr>
</tbody>
</table>

Figure 2 reports the overview of our estimation strategy. The data of previous releases are used for training the prediction model, which is applied to the current (under development) release.

Figure 3 reports the MIND architecture. MIND consists of five main components:

- **Redmine Reader** imports data from a specific Redmine repository and counts the number of defects in a specific class and in a specific release. This component is also in charge of retrieving the ID of tickets related to development.
- **Git Reader** imports data from a specific Git repository and extracts, for a specific class in a specific release, the following data: Ticket ID, class name, class size, and LOC touched.
- **Sonar Reader** checks the compliance of quality rules over several releases of a project. This is a significant improvement over SonarQube functionalities, as SonarQube can only analyze the current version of a project. Therefore, the Sonar Reader extracts the number of violations in a specific class of a specific release. The number of violations are computed by SonarQube, and then read by MIND, over a spectrum of about 1200 quality rules.
- **Data Analysis** merges information coming from the previous three readers and uses open-source packages, like R and WEKA, to perform data analysis and compute the interest.
- **Graphical User Interface** visualizes the data computed and provided by the Data Analysis components.

### III. THE TOOL MIND

#### A. Overview

MIND (ManagIng techNical Debt) works as plug-in of SonarQube and it has been written in Java (7K LOC) over a six month period. MIND is Apache licensed, hosted in SourceForge.net, and is versioned using Git, and tracked in Redmine. MIND supports the analysis of data stored in Git and Redmine and supports all the programming languages supported by SonarQube.

MIND is publicly available in the following resources:

- [Instruction and Demo](http://goo.gl/Ydjtkq)
- [Installation files](http://goo.gl/VvDv75)
- [Redmine repository](http://goo.gl/4r7Y1a)
- [Source files (Git)](http://goo.gl/TWRnat)

#### B. User Stories

MIND supports the following user story: “As a developer I want to know the interest I will have if I will not apply any refactoring so that I can decide if refactoring activities are worthwhile.” A similar, and supported, user story is “As a developer, I want to know the interest I paid, whether increased or decreased over time, so that I can do root-cause analysis.”
Further user stories supported by MIND but not reported in this paper due to space constraints and lack of validation are “As a developer, I want to know which rule, when violated, provides the highest interest to pay so that I can avoid violating it or I can refactor when violated” and “As a developer, I want to know which classes (among the ones under development), if refactored will reduce the highest interest, i.e., where refactoring provides the maximum ROI.”

C. Graphical User Interface

Figure 4 reports a screenshot of MIND visualization of the interest incurred in a software project in terms of how many extra defects occurred, or will occur, due to quality rules violations. MIND reports several metrics related to the interest.

The top-left side of Figure 4 shows that MIND reports the average relative extra defects proneness among classes of the current release (see Equation 6). This metric, in a specific version of the Silverpeas project turned out to be 3.6%. In other words, the current release will have 3.6% more defects if violations are not fixed via refactoring. This information is useful for understanding the benefits the project will have in refactoring the entire software (i.e., paying the principal). In the provided example, a user could decide that no refactoring is required because this will likely result in decrement of only 3.6% the number of defects in the current release. Similarly to the maximum extra defects per 100 LOC touched, MIND reports the relative extra defect proneness as a clickable hyperlink to the list of current classes ranked, in descending order, according to the relative extra defect proneness. This list suggests to the users the classes to which they should perform refactoring activities to reduce the highest relative number of extra defects, regardless of the LOC that will be touched.

The top-middle side of Figure 4 shows that MIND reports the maximum extra defects per 100 LOC touched among classes of the current release (see Equation 4). This metric, in a specific version of the Silverpeas project turned out to be 0.1. In other words, the class more prone to defects, in this release, will probably have one extra defect per 1,000 LOC touched. This information is useful for identifying which class is more defect prone and hence more in need of refactoring, regardless of the amount of LOC that will be touched. The user can also decide if any refactoring is necessary. In the provided example, a user could decide that no refactoring is required because this will decrement of only 1 defect per 1,000 LOC touched in the class more in need of it. MIND reports the maximum extra defects per 100 LOC touched as a clickable hyperlink to the list of current classes ranked, in descending order, according to the extra defects per 100 LOC. This list suggests to the users the classes to which they should perform refactoring activities to reduce the highest number of extra defects, regardless of the LOC that will be touched.

The top-right side of Figure 4 shows that MIND reports the maximum defects per 100 LOC touched among classes of the current release (see Equation 4). This metric, in a specific version of the Silverpeas project, turned out to be 0.2. In other words, the class more prone to defects, in this release, will probably have two defects per 1,000 LOC touched. MIND reports the maximum defects per 100 LOC touched as a clickable hyperlink to the list of current classes ranked, in descending order. This list suggests to the users which classes are more defect prone and hence more in need of refactoring and testing, regardless of the LOC that will be touched.

The bottom-left side of Figure 4 shows that MIND reports the Linkage of a dataset (see Equation 8). This metric, on average across the several Silverpeas releases analyzed, turned out to be 72%. In general, linking commits with tickets is a standard best practice, and the more it is applied the higher the accuracy of MIND. Therefore, linkage information is useful for interpreting the suitability of MIND to analyze a given project. In the provided example, a user could decide to use MIND in the project because a linkage of 72%, although not optimal, is high.

The bottom-right side of Figure 4 shows that MIND reports the mean absolute error of a dataset, on average among different classes of different releases (see Equation 9). This metric, on average across the several Silverpeas releases analyzed, turned out to be 0.00. In other words, there is no error in estimating the interest of a class. This information is useful for understanding the accuracy of MIND output. In the provided example, a user could use MIND output to decide how much the interest should be reduced and in which class. If the mean absolute error of a dataset is high, the user could decide to improve the linkage and the size of the dataset upon which the prediction model is trained and validated.

![Figure 4: Snapshot of MIND’s visualization of the interest.](Image 325x280 to 565x404)

Figure 5 reports a screenshot of MIND visualization of the trend of the interest over time. The horizontal axis reports the time related to a given release (vertical bars below the axis) and the time related to the static analysis of the system (vertical bars above the axis). The vertical axis reports the average relative extra defect proneness. The user can click on a specific time and MIND outputs, in the top-middle, the relative amount of extra defects that occurred at that time in the classes due to violations. In the example reported in Figure 5, the user clicked to an analysis between version 2.0 and 2.1. In that specific moment, the system had 90% more defects due to violations.
In this paper we validate MIND in terms of accuracy in measuring the interest. This translates to compute the error that MIND does in estimating the extra defect proneness.

A. Dataset
We validated MIND over an open-source tool called Silverpeas, a collaborative and social web portal useful for building an intranet and extranet. Silverpeas offers about 30 different applications that permit the user to manage and organize a large variety of contents.

We selected Silverpeas over other open-source projects because it is versioned using Git and tracked in Redmine. Most importantly, Silverpeas has a good level of linkage (e.g., 70%) and its repository is of medium size. Specifically, the Silverpeas repository contains data about 302 different classes and 32 releases for a total of 1.2 million LOC. We analyzed this code with MIND and this resulted in a dataset of 7835 data points. Each data point is a class, and its characteristics, in a specific release (see Table 1).

B. Methodology
MIND accuracy has been computed by measuring the mean absolute error via a 10-fold cross-validation over the Silverpeas dataset.

The adoption of cross-validation is a standard approach for assessing prediction models to obtain realistic and generalizable results. The underlying principle of cross-validation is to modify the dataset on which the prediction is computed by randomly resampling the whole dataset.

We used 10-fold among the several methods for cross-validation. In this approach, the dataset is divided randomly into 10 subparts, constrained to have (approximately) the same cardinality and proportion of categories of the dataset. Each subpart, in turn, is taken away from the dataset, and the remaining data are used to train the model which is then applied to the subtracted part. Finally, performance is computed by averaging the performances that resulted from the ten-step procedure.

To implement cross-validation, we used WEKA because it provides an easy, user-friendly environment to enact cross-validation which allows the user to select the desired amount of folds (e.g. 10). WEKA takes as input the dataset as oracle. Once the type of validation has been defined, WEKA outputs the cross-validated scores of the accuracy metrics [16].

Among the several accuracy metrics, we adopt the mean absolute error (see Equation 9) because it reliably measures the quantity of the error we expect.

C. Results
Figure 6 reports the distribution of the mean absolute error for each observation produced in each fold of the validation. According to Figure 6, the mean absolute error is equal to 0 except very few cases. Moreover, the average mean absolute error turned out to be 0. These results clearly suggest that MIND can accurately measure the interest of a software project with a medium-size dataset.

IV. VALIDATION
We measured the time that MIND required to analyze a Silverpeas release of 56,000 LOC. MIND took a total of four minutes. SonarQube requested half of this time to statically analyze the project over about 1200 quality rules. Therefore, we conclude that the time requested to use MIND is negligible, especially if we consider that the analysis can be scheduled to perform at night.

V. RELATED WORK
A. The importance of the interest
Falessi et al. [8] reported examples, practical considerations, and challenges in dealing with technical debt. Specifically, our conversations with architects, developers, testers, and project managers have revealed a large consensus about that all projects have some technical debt because no project is perfect. In other words, we observed a balancing act: the acceptance of some technical debt which eventually paved the way to MIND development.
Lim et al. [7] reported an interview study, involving 35 practitioners, aimed to characterize technical debt to find out how software practitioners perceive it. Their analysis paints a picture of a complex balancing act of various short- and long-term concerns.

B. Tools for managing Technical Debt

Letouzey and Ilkiewicz [5] proposed the SQALE (Software Quality Assessment based on Lifecycle Expectations) method which is supported by a commercial plug-in for SonarQube. The SQALE Quality Model can be regarded as a projection of the ISO 9126 model and is based on eight different characteristics: reusability, portability, maintainability, security, efficiency, changeability, reliability, and testability. In SQALE there is no explicit reference to the principal and interest components of the debt. In SQALE the term technical debt refers implicitly to the principal which is also called remediation cost. The interest is measured by SQALE in a metric called Business Impact which is in the range \([0, \infty]\). The peculiarity of the Business Impact is to capture eight different aspects of the interest (i.e., the ISO 9126 model characteristics) however it may lack of practical meaning. The main disadvantage of MIND, compared to the Business Impact, is that MIND measures only one characteristic of the interest (i.e., extra defect proneness); however, this characteristic has practical meaning.

Kurtis et al. [17] reported a study summarizing the results of analyzing 745 business applications via the CAST’s Application Intelligence Platform (AIP). They propose and assess a formula for estimating TD with adjustable parameters.

Vallary et al. [9] presented a method for the measurement of TD interest in terms of extra comprehension effort. We share with them the motivation of trying to measure the interest because this information is, at least, as important as principal information. Differences between this study and Vallary et al. [9] include:

- They measure the interest in terms of extra comprehension effort whereas we measure it in terms of extra defects. These two aspects of the interest are clearly different and not overlapping. We believe that long-term solutions, for monitor the interest, must support both of them.
- Their approach uses two quality rules (i.e., two code smells identified via five metrics) whereas we use the 1200 quality rules supported by SonarQube.
- Their focus is on proving the feasibility of the approach whereas our focus is on presenting a working tool and its statistical evaluation.
- They use as source of information an external tool (Blaze) which counts comprehension effort (or proxy measure). MIND uses as source of information the tools already in use, like the issue tracking system and the control version system. Our solution is clearly more easy to use because it does not require any external installation other than the plug-in itself but it requires specific technology to be in use (i.e., Git and Redmine).

Siebra et al. [18] investigated the use of a subset of automatically collected metrics to identify and monitor TD over the course of 7 years on a real project. Their approach links specific metrics to attributes of TD, and hence focuses more on principal than on the interest.

C. Defect Prediction

Defect prediction (i.e., the estimation of where faults are likely to occur in the code) supports test effort, reduces costs, and improves the quality of software [11]. Moreover, if the prediction model produces accurate estimations, we can conclude that predictor (input) variables are correlated with predicted (output) variables.

Hall et al. [11] reported a systematic literature on studies about the performances of prediction models. Their results show that simple techniques outperform complex ones. This is the reason why MIND uses Linear Regression as prediction model.

D’Ambros et al. [19] evaluated the performance of different defect prediction approaches using different performance indicators. Their results indicate that some approaches are statistically better than other and that generalizing results to different contexts/learners proved to be partially unsuccessful.

VI. Conclusion

In this paper we presented MIND: an open-source tool which is, to the best of our knowledge, the first tool supporting the quantification and visualization of the interest of technical debt. By analyzing historical data coming from Redmine and Git repositories, MIND reports the interest incurring in a software project in terms of how many extra defects occurred, or will occur, due to quality rules violations. This information can support the user’s understanding of how much and why the principal should be reduced.

We evaluated MIND by using it to analyze a software project stored in a dataset of more than a million LOC. Results suggest that MIND accurately measures the interest of technical debt.

We plan to improve MIND, and the underlying methodology, in the following aspects:

- **Quantifying the effect of technical debt on development speed.** MIND currently quantifies the interest (i.e., impact of high violation density) in terms of extra defect proneness. However, we expect that high violation density also leads to extra development effort. Therefore, we plan to verify and develop a feature in MIND for quantifying the interest in terms of extra effort. Specifically, we plan to measure the decrement in development speed, that occurred (or will occur) due to violations.
- **Automatic ranking of quality rules.** One of the main problems when deciding which violation to remove via
refactoring is the high number of rules and their blurred importance. For example, there could be a case in which two sets of violations (e.g., 5 Critical and 10 Info) have the same principal, (i.e., they require the same effort to be removed). In this example, the developer does not know which set to remove (i.e., which has the highest interest) for minimizing technical debt. Moreover, the impact of violations can change not only across rules but also across contexts. Thus, without a context-specific validation and prioritization of quality rules, developers cannot effectively manage technical debt. Therefore, we plan to verify and develop a feature in MIND for automatically ranking quality rules. Specifically, we plan to compute the correlation between violation density of a given rule and defect injection frequency. This correlation will differ among quality rules and act as a reliable proxy for their importance.

- **Predicting LOC that will change.** In order to support the developer in deciding which class to refactor, it is important to identify which class will be touched most. We note that, in theory, if a class will not be touched, it has no technical debt; this is because the class will not negatively influence the software project. Thus, developers should refactor only classes that will be touched. Therefore, we plan to verify and develop a feature in MIND for estimating the LOC that will be touched in a class. We plan to implement this feature by analyzing, via natural language techniques, the text of the requirements assigned to future releases.

- **Supporting more technologies.** MIND can be used on projects tracked with Redmine and versioned with Git. In order to improve the spectrum of projects upon which MIND can be used, we plan to enhance the number of technologies supported by MIND. Specifically, we plan to support more Issue Tracking Systems (e.g., Jira and Rally) and more Version Control Systems (e.g., SVN).

- **Adopting more accurate prediction models.** MIND currently adopts Linear Regression as the prediction model to estimate the extra defect proneness. The presented results showed that MIND accurately measures the interest of technical debt in a project that has high linkage and dataset size. We expect MIND accuracy to decrease when linkage or dataset size decreases. In order to make MIND accurate to a larger spectrum of projects, we plan to use more accurate prediction models. Specifically, we plan to compare the accuracy of different prediction models in estimating the interest of technical debt. Afterward, we will incorporate in MIND the prediction model featuring the highest accuracy.

- **Adopting more predictor variables.** MIND currently estimates defect proneness according to violation density only. Again, in order to make MIND accurate to a larger spectrum of projects, we plan to use predictor variables other than violation density. Good candidates for predictor variables are: number of commits, number of bugs in previous releases, and number of developers.

**ACKNOWLEDGEMENT**

This work has been supported by Cisco research grant CG# 583194. This work has been partially performed when both co-authors worked for the Fraunhofer Center for Experimental Software Engineering (CESE), USA.

**REFERENCES**


