sP2D2: Software Productivity and Popularity of Open Source Projects based on Defect Technical Debt

Michael T. Shrove  
Millennium Corporation  
Huntsville, AL. USA  
tshrove@gmail.com

Dr. Emil Jovanov  
University of Alabama Huntsville  
Huntsville, AL. USA  
emil.jovanov@uah.edu

Abstract—The goal of this research is to help software product owners and decision makers predict early in the software project when or if the project is at risk of failure. If the decision makers can get an early notification of the outcome, they can make better choices on what they need to do to make the project successful. Particularly, open source projects present a different kind of problem because they are continuously being changed and do not necessarily have an end to the project. In addition, their determination of success and failure are different than traditional software projects with predefined success criteria. In this paper, we explore creating a likelihood of success factor for open source projects to use for determining the success of their projects. We propose use of number of bugs as technical debt (TD), the productivity element, and the GitHub StarGazer (SG) metrics as the popularity element. We use publicly available TD and SG data to calculate the likelihood of success rate of open source projects using our novel algorithm called Software Productivity and Popularity of Open Source Projects based on Defect Technical Debt (sP2D2). This paper presents an implementation of the sP2D2 method and results of analysis of several open source GitHub projects.

Keywords—Agile Software Development, Project Management, Project Outcome Prediction, Technical Debt, Open Source, Productivity, Popularity

I. INTRODUCTION

In 1992, Cunningham first introduced the idea of Technical Debt (TD) [19] as a concept in software development that reflects the implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer. TD represents the cost of shortcuts in the development efforts of a project according to the customer’s request before the deadline but does not quite produce the code good enough to have a long-term solution, which may lead to unmanageable projects. He also explained that “excess quantities of TD will make a program unmasterable, leading to extreme specialization of programmers and finally an inflexible product.” [19] TD also creates TD interest, as the time after writing not-quite-right code until the TD is rewritten to be properly developed. [19] Every day, company’s development teams or in this case, open source development teams have to decide what to implement and how the code is going to be written. These decisions are what ultimately introduces TD into the project more so in open source software projects. In fact, open source development teams may have more problems with TD than others. They don’t necessarily have a deadline; in turn this may lead to developers never paying back the debt from the development efforts they contributed to. Moreover, open source development teams cannot completely eliminate TD [13], because it could always be introduced into the software project by decisions from the product owners (new goals) or the development team (deadlines or poor decisions). Because of how open source projects are managed, they don’t necessarily have someone monitoring TD and reprioritizing the backlog. The only way to deal with TD is to manage it [15], but manage it in an open source manner where everyone can see the total TD and the effects of the TD on the users. However, managing TD requires many steps in order to do it effectively, such as:

- **Identify the TD.** What types of TD does the team have? Where do these TD items live in my project? Is the TD process related, security related, UI related, or code related?
- **Track the TD.** What software does the team use to track TD? Does the team use multiple tools? Will we have to develop our own tools to track TD?
- **Prioritize the TD.** What TD is most important to deal with? What TD is least important? How will the team prioritize the TD (e.g. work effort, cost, length to complete).
- **Monitor and Visualize the TD.** How will the team visualize the TD and how will they monitor it? Will they use a backlog? A line graph?
- **Acting on the TD.** Does the team have a process in place to deal with high priority TD? Does the team have a process in place to deal with large quantities of TD?

In [16], authors concluded that only about 27% people identified as not having some mechanism or process for identifying TD. The teams, that did not identify TD, either waited for something to fail first and then identified TD as the cause or waited until the schedule exploded before identifying the TD. Furthermore, of the teams that did identify TD, they used either issue tracking systems (e.g., Atlassian JIRA, Gitlab) or social processes (e.g., retrospectives, architecture evaluations). Zazworka in [14], talks about 4 different approaches for identifying TD: modularity violations, design patterns and grime buildup, code smells, and ASA issues. Both [17][13] describe a simple method for estimating the principal and interest of TD items. Furthermore, [13] also describes a method for identifying, measuring, and monitoring TD through...
a software project. In this research, we focused on taking the estimated TD principal along with a usage metric and using those values to produce a real-time visualization to the development community so they can have an idea of the likelihood of the outcome (successful or failure) of a software project based on actual user metrics and TD. In Section 2, we present overview of the two inputs to our algorithm: TD and GitHub StarGazer (SG) and the foundational knowledge needed to understand the research. In Section 3-7, we describe our proposed approach, algorithms, libraries, and methods used. Section 8 presents the result and analysis of the implemented method.

II. OVERVIEW OF TECHNICAL DEBT AND GitHub StarGazer

In this section we will cover the foundational knowledge needed to understand the rest of the paper. In section 2a, we will discuss technical debt. Lastly in section 2c, we will cover our usage metric, the GitHub StarGazer metric.

A. Overview of Technical Debt

As described in the introduction, TD is described as shortcuts team members (i.e. developers, testers, software managers) take to gain business value [17] in the short term, while pushing off those consequences incurred to a later time in the project. Every time TD is incurred in the project, it creates more TD cost [17][20] that has to be monitored. If these costs are not dealt with, they can accrue interest [17][20] on top of the principal [20] cost of the TD item [20]. Recurring TD costs or TD that the team members have decided to not deal with can cause the project to bankrupt [20], causing the project to fail. Failure is what the goal of this research is trying to prevent in order to achieve higher rates of success which in turn means more product downloads or increased usage metrics. In order to be more successful with monitoring TD, we first have to identify the TD in a project. In this research we are going to focus on 5 TD categories: Code Debt [20][18], Defect Debt [20][18], Process Debt [18], Security Debt, UI Design Debt, and Requirements Debt [20][18].

B. Categories of Technical Debt

Within every software project, there is TD. Each TD item is categorized by the area it affects the most. Two studies [20][18] survey TD categories. In this section, we will talk about a sub collection of these TD categories relevant to our study and discuss each TD item category and its applicability.

1) Code Debt refers to any problems in code that can cause maintenance problems later in the project. A few examples include code that was poorly written [18], code without comments, code that does not follow coding standards [20][18], code duplication [20], or code complexity [20].

2) Defect Debt refers to any defects or problems found in the software during testing activities [20], during the testing phase, or defects reported by the users [18]. These are commonly called bugs by practitioners. Most of the time these debt items are tracked using a defect tracking system (e.g. JIRA, Redmine, Bugzilla) and prioritized by severity levels defined by the team.

3) Process Debt [18] refers to any gaps or artifacts missing between the initial definition of the process and actual implementation. An example would be that the process that was defined is no longer applicable or up-to-date and has yet to be updated.

We also introduce two new categories of TD:

4) Security Debt refers to code that may have caused or has the potential to cause a security risk in the software (vulnerabilities). These types of debt can be identified using a static code analysis or dynamic code analysis tools (e.g. Fortify). This category of TD debt can have huge interest associated. If a security TD item was to get released and some bad actor (e.g. black hat hacker) was to exploit the vulnerability, it can have huge costs to fix the additional problems caused by the exploitation but also the costs to the users of the software.

5) UI Design Debt refers to the deficit between the optimal user interfaces (UI) for the perfect user experience (UX) [8] and the actual implemented UI by the software team. UX can be another TD item that has a high interest rate. Usually, software projects will fail in this area because the user experience is not optimal enough for the user, causing rewrites to match the customers’ expectations.

C. Overview of GitHub StarGazer

Today, GitHub is one of the most popular source code repositories. It was founded by a company called Logical Awesome from San Francisco, CA in 2008 [9]. By 2013, it had over 3 million subscribed users to the platform and in 2018, it was acquired by Microsoft [10]. Today it has over 31 million users subscribed. GitHub was based on the popular Git version control technology. GitHub incorporates not only source code storage but also limited social features such as ways to follow other developers or “star” or “gaze” a project [2]. This gaze feature allows developers to like certain projects and determine popularity and usefulness by looking at a project’s stargazer points. The number of StarGazer (SG) points a project has is a direct comparison of the popularity of the project [5]. In [5], The authors proposed a popularity metric calculated by the following:

\[\text{Popularity} = (#\text{Stars}) + (#\text{Forks}) + (#\text{Pulls}) \]  

In this research, we focused on just the # of stars in each repository.

\[\text{Popularity} = (#\text{Stars}) \]  

Later, we will describe in more details the algorithms and how we used the SG metric.

III. PROPOSED APPROACH

Since the publicly available information, such as effort and actual cost, is not available for open source projects, use propose the use of bugs as technical debt (TD), the productivity element, and the GitHub StarGazer (SG) metrics as the popularity element. We use publicly available TD and SG data to calculate the likelihood of success rate of the open source projects using our novel algorithm called Software Productivity and Popularity of Open Source Projects based on Defect
Technical Debt (sP2D2). Figure 1a represents a theoretical open source project which we classify as a bad project. We describe it as bad because the number of defects continues to increase causing TD to rise which in turn causes open source developers to stop using the project and which causes a decrease in SG points. Looking at Figure 1a, the red line representing the cumulative sum of TD cost [20][17] accumulated over time (error, bugs, etc. [20]) at time t over the project’s lifespan. Time 1.0 represents the most current time in the project. Each point can represent a time in the project at which a developer removed TD. The blue line represents the cumulative sum of SG points of the repository associated with the open source project. The green line on the bottom graph of Figure 1a, represents what a theoretical sP2D2 rate would look like based on the top TD and SG data. As you can see at time 0.5 (halfway between the start of the project and the most current time), the TD started to be increase and the SG rate started to decrease. Therefore, the sP2D2 rate started to show a decrease (downward slope) in the likelihood of the project succeeding. Similarly, Figure 1b shows what the failing project would look like. For a fictional open source project that we would classify as a good project at time 0.5 the TD (red line) starts to decrease and the SG rate (blue line) starts to increase. Therefore, our sP2D2 rate shows an increase (upward slope) indicating the likelihood of the project succeeding. Product owners of open source projects can use this to monitor and set their own thresholds using our sP2D2 graph. These thresholds can then be used to trigger certain events in order to mitigate the decrease in likelihood of success.

IV. The Datasets

We used two different datasets in order to achieve our research objective. Both datasets are from a company called MongoDB. Both were readily available and open to the public. The first dataset used was from MongoDB’s JIRA API [23]. MongoDB has been tracking a set of their software projects over the past 9 years using Atlassian JIRA as their issue tracking system. Their projects are mostly the drivers used for connecting various programming languages to the MongoDB platform. There was a total of 31 projects. Out of the 31 projects, we chose 7 for our research. Out of the 7 projects, the mean lifespan was 9.64 years ± 172 days. The shortest project current has a lifespan of 3,286 days and the longest project having a lifespan of 3,679 days. The other dataset was MongoDB’s GitHub repositories. Each project in JIRA had an equivalent GitHub repository. We used the SG data from each repository to determine the popularity over time and get some usage metrics from each one to pair with defect data (TD) from each JIRA project.
We used an open source library called jira-python \cite{22} to extract datasets from MongoDB’s JIRA API and PyGithub \cite{5} for extracting GitHub datasets from MongoDB’s GitHub API \cite{11}.

V. TECHNICAL DEBT MANAGEMENT FRAMEWORK

A TD management framework is key to keeping TD \cite{14} identified and \cite{13} measured. In order to keep a project successfully on track, all TD has to be identified not only the visible TD (e.g. code debt \cite{18} and requirements debt \cite{18}) but also invisible debt (e.g. architecture debt \cite{18}, security debt). In Section 2B, a few TD categories were identified, however we decide to only use defect debt in our research. Open source project tends to not have extensive identification and separation of their TD by specific categories. Open source projects mainly rely on defect debt as the most used TD category. In the next section, we will discuss in details of how these categories of TD were identified and what technique we used for TD cost.

A. Identification of Technical Debt

Identification of the TD items in the JIRA dataset presented in Section 4 was very straightforward. MongoDB had done a very good at identifying and labeling defect debt. For every defect debt, they had created a Bug issue type in JIRA. The main focus of this research was to identify, measure and use TD in determining whether an open source project was successful or not at time $t$.

B. Prioritization Measurement Approach of Technical Debt (PMAT)

In \cite{1}, the layout the research describes is a straightforward path for getting to the total TD impact or in our case, the cost of the TD over the project is currently unresolved. First, we identify the TD. Secondly, we measured the principal and the interest of the TD item. Lastly, we combined those two to get total TD cost. We used an automated estimate technique \cite{1} in our experiment to measure TD because we wanted an automated mechanism that would identify, measure, and produce the results for the product owners without human interaction. We also did not measure the interest of each TD item because we were more interested in getting the principal of each TD item. Looking through the JIRA dataset collected from MongoDB’s JIRA, we ended up calculating the TD principal using a normalized value. Each issue in the MongoDB JIRA dataset was assigned a priority. MongoDB’s priority levels were predefined. The priority levels range from one to five as seen in Table 1.

<table>
<thead>
<tr>
<th>Priority Level Name</th>
<th>Priority Level Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trivial</td>
<td>1</td>
</tr>
<tr>
<td>Minor</td>
<td>2</td>
</tr>
<tr>
<td>Major</td>
<td>3</td>
</tr>
<tr>
<td>Critical</td>
<td>4</td>
</tr>
<tr>
<td>Blocker</td>
<td>5</td>
</tr>
</tbody>
</table>

Employing the knowledge in what we had learned from the dataset, we arranged the TD items in order by date, starting with the earliest in the project to oldest in the project for each project in the JIRA dataset. Using the priority levels defined, a TD principal was assigned to each TD item associated with the TD items priority value as defined by Table 1.

As seen in Figure 3, Time 0 is the first time in the project that a TD item was entered into the issue repository. Each subsequent time after that is a timestamp of when a TD item was entered into the issue repository associated with a TD principal. The timestamp also could be associated with a resolution of a TD item (e.g. -5, -2). Both situations we call a TD item event. After creating a list of ordered TD item events (positive value for new TD or negative value for resolution of a TD item) we stored them into an array. We chose to use the percentage completed of the project for normalization of all TD items in the project from the dataset, no matter the time range of the project. Lastly, we performed a cumulative sum of the TD item principal array. After getting a cumulative summation of all TD items, we performed an interpolation on the array in order to reduce noise and smooth the signal out. These values would be the y-value for the first dataset into our sP2D2 algorithm. Next, we need to perform a normalization on each TD item’s time value. The goal was to have the first value be at time 0.0 in the project and the last value be at time 1.0. Each value between the first and the last would be a ratio of the project’s max time. These normalized values would be the x-value for the first dataset into our sP2D2 algorithm.

VI. STARGAZER MANAGEMENT FRAMEWORK

Similar to the TD management framework, the SG management framework needed to perform basic extract, transform, and load (ETL) functions. We had to first take the names of the JIRA projects identified and also identify associated GitHub projects. That is why we decided to use the MongoDB datasets, because they presented both JIRA projects that were open and available along with GitHub projects \cite{11} that were open and available.

A. StarGazer ETL Engine

First, we had to collect the raw data from GitHub. We decided to use PyGithub \cite{1}, an open source python library used to connect and work with the GitHub API v3. Once we
connected to the database, we pulled all SG data for each project.

Second, we had to transform the data into the correct format for our research. The GitHub SG data was given to us in JSON format via the API. We extract each field needed (value, date) and stored it into a MySQL database.

Lastly, we used the time stamps and normalized them for the project’s lifespan. 0.0 being the start of the project and 1.0 being the most current data of the total project. In order to do this, we had to find the minimum date from both the JIRA project data (TD data) and the GitHub SG data in order to find our true start date of the project as seen by equation 4. Likewise, we had to find the project most current data by finding the maximum of both the GitHub SG data and the TD data. A representation of the workflow can be seen by Figure 2.

\[
\alpha = \max(\max(TD), \max(SG)) \quad (3)
\]

\[
\beta = \min(\min(TD), \min(SG)) \quad (4)
\]

VII. SP2D2 ALGORITHM

Primary objective of our algorithm is to show the product owner when the likelihood of success of the project was either increasing or decreasing. To do this we took both the GitHub SG data (normalized and interpolated) and the JIRA TD data (normalized and interpolated) and these were the inputs into the algorithm. Next, we calculated the percentage change for each item in both sets of data.

\[
SGP_i = \frac{(SG_i-SG_{i-1})}{SG_{i-1}} \text{ for } i = 2 \text{ to } \text{length}(SG) \quad (5)
\]

\[
TDP_i = \frac{(TD_i-TD_{i-1})}{TD_{i-1}} \text{ for } i = 2 \text{ to } \text{length}(TD) \quad (6)
\]

Next, we took both outputs from the equation 5 and 6 and subtract, SGP2 from TDP2 to get a singular output array. As you can see in Figure 2, the output from our sP2D2 algorithm was the values graphed for our final results.

VIII. RESULTS AND DISCUSSION

In this section, we will discuss the results of our research. We applied these algorithms and ideas to the two different projects from MongoDB project sets. The results are described below.

A. TD and SG Management Framework Results

After we had gathered the necessary data from MongoDB’s JIRA and GitHub APIs, we were able to apply the algorithms discussed in this paper to their data. The first task we performed was to choose some existing projects. We decided on two projects that were currently in progress: PERL and CSHARP. Both projects are connectors (middleware) used for the PERL and C# languages to connect to the MongoDB platform.

Fig. 2 - High Level View of Workflow of Data
A steady decrease in the graph. However, if the SG continues to increase over time without a major decrease in the overall TD. The expected results from our sP2D2 visualization should show a steady decrease in the graph until the 0.5 mark at which the graph should start to increase because of the decrease in TD. As you can see in Figure 1d that is exactly what is shown. We show a gradual decrease in likelihood of success until the 0.5 mark at which TD decrease showing an increase in likelihood of success shown by our sP2D2 graph.

**IX. CONCLUSION**

Open source software projects need to continuously monitor software projects and identify as early as possible in their project that the potential for failure or a decrease in usage of the project will occur. In this research, we propose the use of bugs as technical debt (TD), the productivity element, and the GitHub StarGazer (SG) metrics as the popularity element. We use publicly available TD and SG data to calculate the likelihood of success rate of open source projects using our novel algorithm called Software Productivity and Popularity of Open Source Projects based on Defect Technical Debt (sP2D2).

Analysis of the set of real software projects indicate that the proposed method can effectively be used to show decrease and increase in likelihood of success. We believe that our algorithm can be further improved by using more accurate TD measurements and the use of TD interest. TD interest can ultimately cause smaller principal TD events to result in higher TD costs over time. We plan to use regression analysis on both the SG and TD datasets in order to forecast the success likelihood in the future.

**REFERENCES**


