Journal of Software Engineering Research and Development, 2023, 11:1, doi: 10.5753/jserd.2023.2417

Technical Debt Guild: managing technical debt from code up to build

Thober Detofeno [Pontifícia Universidade Católica do Paraná | thober@gmail.com]
Andreia Malucelli [Pontifícia Universidade Católica do Paraná | malu@ppgia.pucpr.br]
Sheila Reinehr [Pontifícia Universidade Católica do Paraná | sheila.reinehr@pucpr.br]

Abstract

Efficient Technical Debt Management (TDM) requires specialized guidance so that decisions taken are oriented to add value to the business. Because it is a complex problem that involves several variables, TDM requires a systemic look that considers professionals' experiences from different specialties. Guilds have been a means technology companies have united specialized professionals around a common interest, especially those using the Spotify methodology. This paper presents the experience of implementing a guild to support TDM's activities in a software development organization using the action research method. The project lasted three years, and approximately 120 developers were involved in updating about 63,300 source-code files, 2,314 test cases, 2,097 automated test scripts, and the build pipeline. The actions resulting from the TDM guild's efforts impacted the company's culture by introducing new software development practices and standards. Besides, they positively influenced the quality of the artifacts delivered by the developers. This study shows that, as the company acquires maturity in TDM, it increases the need for professionals dedicated to TDM's activities.

Keywords: *Technical Debt, Technical Debt Management, Community of Practice, Technical Debt Guild*

1 Introduction

Technical Debt (TD) is a metaphor that expresses software artifacts' immaturity and their impacts on software maintenance and evolution activities. According to Brown et al. (2010), this metaphor characterizes the difference between a software system's current state and its hypothetical ideal state. A theoretical ideal state is understood as the one established by the context in which the software is inserted (Brown *et al.*, 2010). TD negatively affects productivity and feasibility in software development. In many cases, developers are forced to introduce more TD because of prior debts (Besker *et al.*, 2019).

It is estimated that between 25% and 37% of all development time is wasted due to TD. Most of the time is wasted understanding or managing TD (Ampatzoglou *et al.*, 2017; Besker *et al.*, 2017; Martini *et al.*, 2018). If unmanaged, TD can result in significant cost overruns, serious quality problems, reduced developer morale (Ghanbari *et al.*, 2017), and limited ability to add new features (Seaman *et al.*, 2012). It can even reach a crisis point when a vast and expensive refactoring or complete system replacement is needed (Martini *et al.*, 2014).

The efficient management of TD is a little explored area, although it seems to help in quality and productivity during software development (Guo *et al.*, 2016; Rios *et al.*, 2018). Works investigating aspects of TD management in the software development process are isolated initiatives (Rios *et al.*, 2018).

Decision-making in TD management is hard to standardize because, in most cases, it depends on the organization's context (Guo *et al.*, 2016). One way to face this problem is to build a team focused on solving problems. This approach can be a practical way of solving a wide range of issues and offering suggestions on processes and working methods that need improvement (Connolly, 1992). Such groups can be implemented using the concepts of Communities of Practice (CoP) (Smite *et al.*, 2019).

A Community of Practice (CoP) is a group of individuals who periodically meet due to a common interest in learning and applying what has been learned, sharing knowledge, exchanging experiences, taking their problems, and finding solutions. One of the best-known examples of CoP is the concept used by the music streaming technology company Spotify, named guild (Kniberg, 2014).

In a context where the TDM should be incorporated into the software development process, bringing together people who have knowledge and interest in the subject can contribute to finding solutions and generating value for the business.

This article presents an experience report on establishing and using a TD guild in a software development organization throughout an action research process. The paper describes the experiences, results, success factors, and challenges. The actions promoted by the TD guild contributed to the Identification, Monitoring, Prevention, Prioritization, and Payment activities in TDM. The guild helped align TD's payout efforts with the organization's goals.

Due to the several strategies that can be adopted and difficulties in measuring the results, implementing a TDM process is not a trivial task. This work is expected to support other companies in the challenge of TDM using a guild approach.

This study is structured as follows: section 2 presents a literature review; section 3 presents the research method; section 4 describes the context and overview of the company in which the study was conducted; section 5 describes the three cycles of action research; section 6 presents the results, lessons learned, challenges, related work and threats to validity. Finally, section 7 concludes the paper.

2 Background

2.1 Guild or Communities of Practice (CoP)

In the middle age, guilds played an essential role in economic sustainability. A guild was formed hierarchically by masters, officers, and apprentices and had experienced and renowned specialists in its field of craftsmanship. These specialists were called master artisans. There was an exchange of knowledge in these guilds to make the work more efficient and productive (Wolek, 1999).

Using these older phenomena as a reference, Leave and Wenger (1991) coined the term Community of Practice (CoP). In the most current concept, approached by Wenger and Wenger-Trayner (2015), CoPs are formed by people who share a concern or passion for something and engage in collective learning in a shared domain of human effort to do it interacting better regularly.

For Wenger, McDermott, and Snyder (2002), domain, community, and practice are the three essential elements that characterize a CoP. The domain builds the community and identity and corresponds to the interest area that attracts and keeps the members. On the other hand, the community is the central element, composed of individuals and their interactions based on joint learning.

CoPs stand out for managing knowledge assets in organizations, creating value for members and the organization, as a competitiveness tool. It can develop new skills and generate strategic opportunities through innovations (Wenger *et al.*, 2002). It is believed that CoPs are used in organizations of different natures, with other terminologies, such as learning networks, thematic groups, technology clubs, and guilds.

The professional literature on how to scale up agile software development suggests CoPs as a possible solution for learning and knowledge sharing among individuals with similar functions, such as Testers or Scrum Masters (Larman and Vodde, 2010).

Experience from four CoPs at Ericsson shows that success factors include a good topic, a passionate leader, a proper schedule, decision-making authority, openness, tool support, a suitable rhythm, and cross-site participation when needed. The CoPs in Ericsson had three leading roles: to support the agile transformation, be part of the large-scale Scrum implementation, and support continuous improvement. CoPs became a central mechanism behind the success of the large-scale agile implementation in the case organization that helped mitigate some of the most pressing problems of the agile transformation (Paasivaara and Lassenius, 2014).

For Smite et al. (2019), implementing well-functioning communities is not easy. Experiences from Oracle Corporation, UK National Health Service, Hewlett-Packard, Wipro Technologies, Alcatel, and DaimlerChrysler suggest that the cultivation of knowledge culture requires organizational attention, support, and sponsorship for CoPs.

Inspired by CoPs, the guilds in Spotify are designed beyond formal structures and unite members with common interests, whether related to leisure (cycling, photography, or coffee consumption) or engineering (web development, back-end development, C++ engineering, or agile coaching).

Figure 1 presents the five types of members identified by Smite et al. (2020) in the guilds of Spotify, based on the numbers of members registered in the communication channels and engaged in the activities. Similar to Wenger et al. (2002), Smite et al. (2020) identified a group of core members (sponsors and coordinators), active members, and peripheral members (passive members and subscribers). The latter group forms most community members (Smite *et al.*, 2020).

Smite et al. (2020) noticed that individual members' activity levels change over time for several reasons: the coordinator role rotates, some active members become passive and vice versa, and those who change specialization turn into inactive users who merely subscribe the latest news.

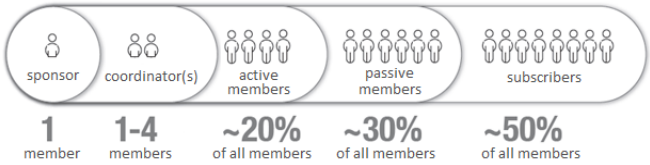

Figure 1. Different types of members in a guild (Smite *et al.*, 2020).

Some guilds arise from shared interests, while others are structured or sponsored and can even have a specific budget. The maintenance and generation of value for the organization of a guild is a challenge.

2.2 Technical Debt Management (TDM)

As previously stated, TD represents the effects of immature artifacts in the software evolution that bring short-term benefits but have to be adjusted later. The concept, whose scope was initially limited to source code and related artifacts, was expanded to consider different software development stages and work products (Alves *et al.*, 2016).

Rios, Mendonça, and Spínola (2018) provide a taxonomy with 15 types of TD, as described below:

- Architecture Debt – "Refers to the problems found in product architecture, which can affect architectural requirements. Usually, architectural debt could result from sub-optimal upfront solutions or sub-optimal solutions as technologies and patterns become superseded, compromising some internal quality aspects, such as maintainability."
- Automation Test Debt – "Refers to the work involved in automating tests of previously developed functionality to support continuous integration and faster development cycles."
- Build Debt – "Refers to issues that make the build task harder and unnecessarily time-consuming."
- Code Debt – "Refers to the problems found in the source code (poorly written code that violates best coding practices or coding rules) that can negatively affect the

[1] https://www.sonarqube.org/
[2] https://github.com/SonarSource/sonar-php

legibility of the code, making it more challenging to maintain".

- Defect Debt – "Refers to known defects, usually identified by testing activities or by the user and reported on bug tracking systems, that the development team agrees that should be fixed but, due to competing priorities and limited resources, have to be deferred to a later time".
- Design Debt – "Refers to debt discovered by analyzing the source code and identifying sound object-oriented design principles violations."
- Documentation Debt – "Refers to the problems found in the software project documentation."
- Infrastructure Debt – "Refers to infrastructure issues that can delay or hinder some development activities if present in the software organization. Such issues negatively affect the team's ability to produce a quality product."
- People Debt – "Refers to issues that can delay or hinder some development activities if present in the software organization". This is represented for late hire, for example.
- Process Debt – "Refers to inefficient processes, e.g., what the process was designed to handle may be no longer appropriate."
- Requirements Debt – "Refers to tradeoffs made concerning what requirements the development team needs to implement or how to implement them. In other words, it refers to the distance between the optimal requirements specification and the actual system implementation."
- Service Debt – "Refers to the inappropriate selection and substitution of web services that lead to a mismatch of the service features and applications' requirements. This kind of debt is relevant for systems with service-oriented architectures."
- Test Debt – "Refers to issues found in testing activities that can affect the quality of those activities."
- Usability Debt – "Refers to inappropriate usability decisions that must be adjusted later."
- Versioning Debt – "Refers to problems in source code versioning, such as unnecessary code forks."

Design, code, and architecture debts are the most studied TD types. This is probably because several source code analysis tools help identify problems such as complex code, code smells, duplicate code, and others, which often serve as indicators of technical debt. The authors also define debt types and a list of situations in which TD items can be found in the software (Rios *et al.*, 2018).

A TD item represents an instance of TD and has several causes - factors that lead to the occurrence of the TD item - and consequences to the project. A TD item can be caused by inappropriate processes, decisions, schedule pressure, etc. On the other hand, TD items can cause several consequences that affect software features and are usually related to cost value, schedule, and quality. A TD item can be associated with one or more artifacts of the software development process (Rios *et al.*, 2018).

If TD items are not managed, they can cause financial and technical problems, increase software maintenance and evolution costs, and lead to a crisis point where the entire future of the software can be compromised (Martini and Bosch, 2016; Spínola *et al.*, 2013; Nord *et al.*, 2012). It is not enough that teams are only aware of what constitutes TD. They must be aligned to manage TD to add value to the business. Simply knowing about TD does not necessarily result in value for the software (Bavani, 2012).

TD metaphor allows thinking about software quality regarding the organization's business (Tom *et al.*, 2013). However, the decision criteria used for the payment of TD can be different according to the different scenarios and objectives of an organization (Rios *et al.*, 2018). A challenge for development teams is to quantify the maintenance problems of their projects to justify the investment in refactoring the TD (Mo *et al.*, 2018; Sharma *et al.*, 2015). Convincing arguments are needed about when and why the TD should be removed.

A model for TDM should foresee the contexts in which TD is identified and evaluated so that decisions can help companies and organizations to take advantage of opportunities and anticipate market needs (Kruchten *et al.*, 2012).

Although TD affects everyone involved in the project, regardless of the cause, the level of communication regarding the TD varies. Team members generally discuss TD among themselves but understand that there are difficulties presenting evidence of TDM to upper-level management (Codabux, 2013).

TDM includes identifying, monitoring, and paying TD items incurred in a system (Griffith *et al.*, 2014). Rios, Mendonça, and Spínola (2018) describe Prevention, Identification, Monitoring, and Payment as macro activities and Documentation and Communication as activities performed during TDM.

Some activities such as identification (e.g., TD detection by static source code analysis), measurement (TD quantification using estimates), and payment (TD resolution by techniques such as re-engineering or refactoring) receive more attention with the support of appropriate tools and approaches (Li *et al.*, 2015).

The payment activity refers to the activities carried out to support decision-making about the most appropriate time to eliminate TD items. At this point, the prioritization of which TD item should be eliminated is made (Rios *et al.*, 2018). The TDM turns it possible to make decisions about eliminating the TD and the most appropriate moment to do so (Guo *et al.*, 2016).

Decision-making criteria are the basis for generating prioritization in the payment of TD items. TDM should be based on a rational approach to decision-making, considering planned and potential future development (Schmid, 2013).

[1] https://www.sonarqube.org/
[2] https://github.com/SonarSource/sonar-php

Guo et al. (2016) mentioned that aspects of managing TD in a software development process were little explored. Until the literature search for this work development, no studies reporting experiences of applying CoPs or guilds to TDM support were found.

3 Research Method

Considering the characteristics of this study, the research method selected was action research. Action research aims to provide research subjects, participants, and researchers to respond to the problems they experience with greater efficiency and based on a transformative action. The characterization of action research varies from one author to another. However, there is a set of common characteristics (Dick, 2000):

- Act in an existing situation to improve and expand knowledge on the subject.
- To have a cyclical nature, to repeatedly execute a series of steps. The cycle varies according to the author, but it must include the stages of Planning, Action, and Reflection.
- Possess a reflexive nature and a critical reflection on the research process and obtained results.
- It is primarily qualitative, although quantifications are possible in some situations.

In Coughlan and Coghlan (2002), the action research cycle comprises three steps, as illustrated in **Figure 2**:

1. a pre-step: to understand context and purpose.
2. six main steps that relate first to the data and then to the action, as follows:
    - Data Gathering: This can occur through observations, interviews, surveys, and reports, collecting qualitative or quantitative data.
    - Data Feedback: the collected data is submitted to the organization for analysis through reports or feedback meetings.
    - Data Analysis: seeks for each party to contribute with a critical view of the data collected, internal company issues, the conduct of the research, or interaction with the researcher's knowledge.
    - Action Planning: what will be done and the deadline.
    - Implementation: the actions are implemented to promote the planned changes in collaboration with the stakeholders.
    - Evaluation: a reflection of the results expected or not, coming from executing the action, aiming to improve the next cycle.
3. A meta-step to monitor that occurs through all the cycles.

Each cycle leads to another, so continuous planning, implementation, and evaluation occur over time, as illustrated in **Figure 2**.

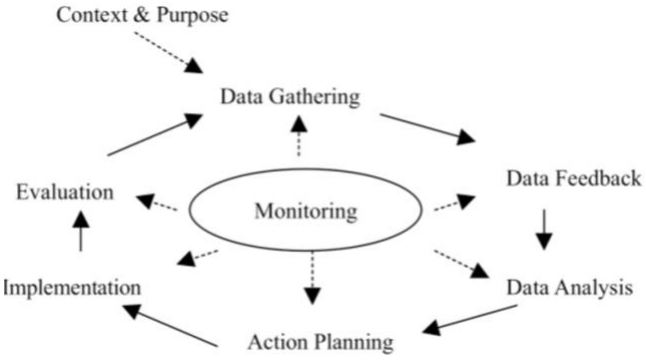

Figure 2. Action research cycle (Coughlan and Coghlan, 2002).

Our study was structured based on this approach, as illustrated in **Figure 3.** It began with a stage of understanding the Context and proceeded with three cycles of the Driving Phase, composed of the following steps:

- Planning: data analysis was performed with those involved to establish what would be done and when.
- Action: the planned activities were implemented to promote the planned changes in collaboration with those involved and responsible for the organization.
- Evaluation: a reflection was performed to analyze the outcomes, aiming to improve the following cycle.

Each cycle of this research was conducted as presented below:

- **1st cycle**: In the first cycle, the guild was created, and the guidelines for the scheduled and unscheduled social interactions were established. The first steps were taken to TD identification, and the teams were guided in the TDs payment and monitoring.
- **2nd cycle**: This cycle was a review of the previous one, where the tools and management activities of TD were revised. The guild promoted the standardization of the source code's development and documentation and guided the teams in prioritizing the TD.
- **3rd cycle**: In the third cycle, the review of the tools and the TD identified in the source code was maintained, and the TD guild sought to identify and propose actions to pay for the TD in the Continuous Integration Test artifacts.

The duration of each cycle is linked to the company's annual management cycle, which foresees periods of planning and execution of actions that impact the software development process or the teams' goals.

[1] https://www.sonarqube.org/
[2] https://github.com/SonarSource/sonar-php

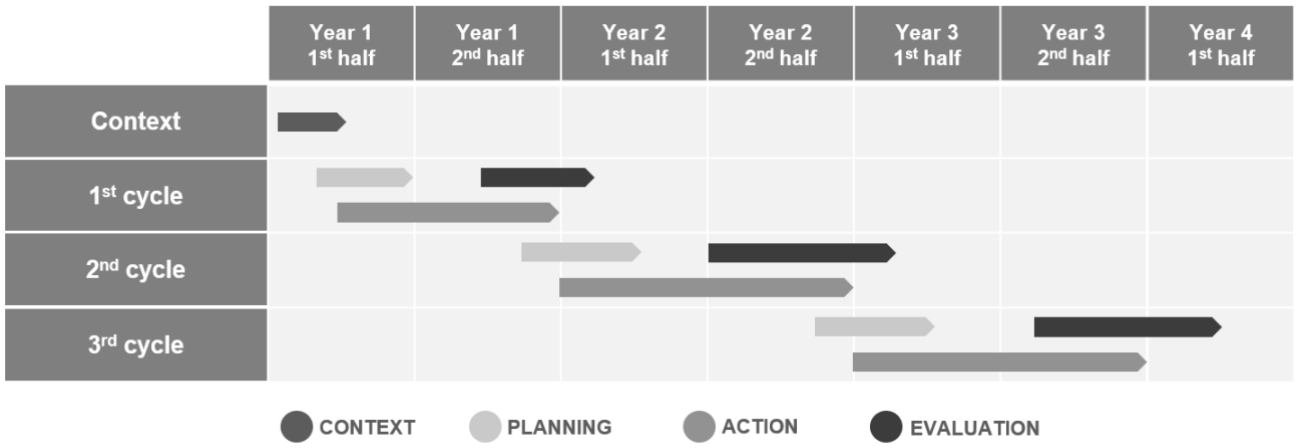

Figure 3. Timeline of the research.

4 Context

This article describes the experience of a TD guild's implementation and evolution in a Brazilian Software Development company founded in 1995. It currently has more than 2,000 customers and 300,000 users worldwide, providing process improvement and compliance management solutions. Corporations use its solutions in all kinds of industries: manufacturing, automotive, food and beverage, mining and metals, oil and gas, high-tech and IT, energy and utilities, government and public sector, financial services, transportation and logistics, and healthcare. Technically the product is entirely on the WEB, with documentation and localization for more than ten languages, and compatible with three database management systems.

The Software Development area brings together some benefits of the agile philosophy with project management. The project management and planning use the SCRUM method defined by Schwaber and Sutherland (2020), dividing it into two-week development cycles and a quarterly release to the market. Thus, the company does not have automated Continuous Delivery or Continuous Deployment. However, it has Continuous Integration with a standardized and automated development flow/process for all teams for software development.

During the period that lasted for three years, the area had, on average, 96 professionals split into 12 teams composed of professionals with the following roles: Product Owners (PO), Scrum Masters (SM), Developers (Dev), Testers, and DevOps. The teams vary in terms of the number of members, the amount of source code they are responsible for, and the programming languages used. The source code repository is composed by 61% PHP, 30% JavaScript, 3% Java, 2% HTML, 2% CSS, 1% JSON and 1% XML. The development area is responsible for approximately 63,300 source-code files. In the second and third years of the study, the monthly average was 1,850 change packages effective in the repository (commits).

TD concepts and TDM activities (as an approach to contribute to quality and productivity during software development) were presented to the Product Owners and Scrum Masters in an internal meeting. The area's director proposed sponsoring and supporting creating a TD guild to discuss and offer TDM solutions for the company.

The invitation for TD Guild was to all professionals involved in the product's maintenance and evolution activities. Per year, three or four experienced professionals were invited to become active members because they had a deep knowledge of the product's architecture.

In the three years that the TD guild was implemented and evolved, the sponsor and the coordinator were the same professional, but there were changes in the active members. **Figure 4** shows the number and type of members per year. In the first year, the active members were composed of a Tester, a Product Owner (PO), a Scrum Master (SM), and three Developers (Devs). In the second year, the members were two Testers, three SM, and two Devs. In the third year, the members were three Testers, an SM, and two Devs. The guild was composed of representatives with technical and business knowledge of the product.

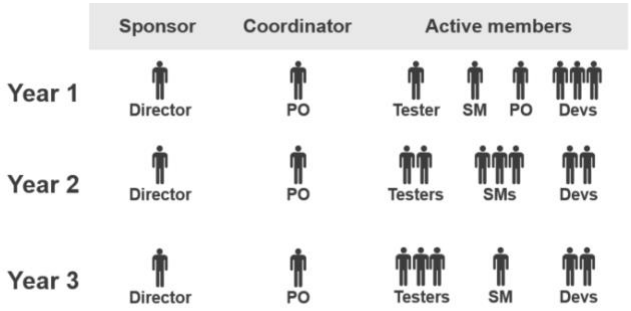

Figure 4. TD guild members.

Most members of the TD guild are peripheral members that do not represent the key practitioners. Peripheral members are those with low involvement in the guild's interactions or the members impacted by the guild's actions. They provided kind suggestions, criticisms or encouraged the initiatives. In the 2nd cycle, these comments were analyzed through a survey.

The TD guild emerged within an organizational context, aligned with strategic objectives, and sponsored by the

[1] https://www.sonarqube.org/
[2] https://github.com/SonarSource/sonar-php

board. Besides the exchange of experiences, learning, and best practices on TDM, the TD guild was challenged to generate value for the product and add knowledge to the software development teams.

The TD guild formation was based on the guidelines for CoPs building and the characteristics of autonomy and alignment with the strategic objectives given by the Spotify approach presented by Kniberg (2014). The guild meetings were monthly and face-to-face, but the members met more frequently to deliberate actions that required more speed in specific cases.

The primary responsibilities of the TD guild coordinator were to organize the subjects and meetings, monitor the execution of tasks, support guild members, and align the needs with the sponsor. The sponsor was responsible for evaluating the proposed actions and approving and providing the necessary resources to execute the tasks.

During guild meetings, each group member presented the ideas and problems to which the guild should pay attention. For each action approved by the guild members and the sponsor, a task list was created with a guild member as responsible. The person in charge had the objective of continuing the theme, carrying out in-depth studies and practical tests to evaluate the proposal's feasibility. The sponsor approved the tasks so the responsible in charge could prioritize this task and the other demands of the team.

The subjects or actions in progress were discussed at the beginning of the guild meetings. The specific issues were often discussed in an internal communication channel or by e-mail.

5 Research cycles

The TD guild beginning was marked by discussions and alignments about the purposes, objectives, guidelines to conduct the activities, and interest subjects to the organization and its members. At the beginning of each research cycle, guild members reviewed goals and procedures.

The TD guild's purpose was to study and help implement and monitor the TDM, with proposals and actions to improve internal quality and reduce maintenance costs and software evolution. To carry out its duty, the TD guild developed some directives to conduct the meetings and activities aligned with the organization's expectations, as follows:

- Be aligned with the company's strategy.
- Have a well-defined purpose or objective.
- Have autonomy to implement solutions.
- Clearly communicate the problems and opportunities to the interested parties.
- The member must be a promoter for TD's payment actions within the teams.
- Allow members of different teams to participate, considering that the member should have knowledge about the work context.
- The member influences the teams to help direct and prioritize the tasks of refactoring the TD.
- Maintain the focus on quality and productivity in software development, helping to define the actions of prioritization and payment of the TD.
- Guide the teams on best practices and standards of internal development.
- Have periodic meetings to monitor the actions and propose changes.

5.1 First cycle

In the first year of this study, the actions of the TD guild focused on two initiatives related to the PHP source code: (1) Identify, measure, and monitor the primary Technical Debts identified in the PHP source code; (2) Identify and propose actions to improve the PHP source code.

Based on the guidelines and the deployment of the initiatives, the TD guild defined the following actions:

- Deploy tools to support TDM.
- Identify TD in the context.
- Guide teams on TD payment.
- Monitor TD payment.

The guild contributed to disseminating the TDM within the company, identifying the most appropriate TD for the company's objectives, and selecting the most appropriate TD identification and monitoring tools. According to the company's goals and resources, the guild members' quality rules priority classification guided the TD payment.

5.1.1 Deploy tools to support TDM

To manage the TD, it is necessary to have tools for implementing and continuing actions. Tools provide support and enable the automation of TDM activities. SonarQube[1] and the SonarPHP[2] plugin were selected to identify and monitor the PHP source code's TD. The choice of SonarQube and the SonarPHP plugin was mainly to the: amount of quality rules available for the PHP source code; options for configuring the quality rules; and, the possibility to develop specific quality rules.

The quality rules provided by SonarQube were reviewed and updated according to the organization's context. The SonarQube identified the source code that was not within the TD guild's coding standard.

5.1.2 Identify TD

In this action, the TD guild's objective was to know in detail and select the quality rules provided by SonarQube considering the company's goals.

Table 1 presents the list of options to select and classify the quality rules. The classification and selection of quality rules were made based on priority definition. The description of priorities was defined by the TD guild, taking into account the organization's context, and were used to guide the classification of the quality rules. The rationale for using this

[1] https://www.sonarqube.org/

[2] https://github.com/SonarSource/sonar-php

scale is to allow easy mapping to the five-point scale used in SonarQube: Blocker, Critical, Major, Minor, and Info.

Table 1. List by priority.

Priority	Description
Blocker	Rule considered as a bug, system vulnerability, or command that should not be used.
Critical	An important rule with a high impact on product quality and source code standardization.
Major	A minor rule with a low impact on product quality
Minor	Good practice rules that should be monitored.

This action resulted in the approval of 93 quality rules that were activated in SonarQube, classified by priority and TD type, as shown in Table 2.

Table 2. First cycle: Rules classified by priority and TD type.

Priority	Blocker	25
	Critical	27
	Major	14
	Minor	27
TD type	Code Debt	39
	Defect Debt	18
	Design Debt	36

5.1.3 Guide the teams in TD payment

The quality rules were classified by their priority, complexity, and impact to support the teams prioritizing and paying the TD. The priority analysis ranked the quality rules considering the research context's available objectives and resources. This analysis was used to prioritize TD payment actions.

The quality rules' analysis on complexity and impact helped the teams select the TD payment source files. Complexity was understood as the technical difficulty to solve a quality rule. The impact of a change was classified by the extent of the change within the system, that is, the change's potential to affect other modules or classes. Some members ranked the quality rules separately, following the guild's guidelines. The results of the classification were reviewed and aligned during the guild meetings.

5.1.4 Monitor TD payment

The TD monitoring was intended to expose the reality and motivate the teams to pay the TD. To support the teams in the periodic monitoring of TD, the SonarQube was configured per team, and a web portal with the values of the TD classifications was made available. This action facilitated the management's follow-up on the teams' initiatives in TD payment.

5.2 Second cycle

At the beginning of the second cycle, the guild members discussed and decided to maintain the objectives and guidelines defined in the first cycle. However, they added the initiative to identify and propose improvement actions in the PHP source code most relevant to the project. Thus, the TD guild defined the following steps:

- Deploy tools to support TDM.
- Define a coding standard in PHP.
- Define a documentation standard in PHP.
- Identify TD in the context.
- Train the teams on the standards and best practices.
- Evaluate guild actions by the developers.

The actions to monitor and guide the teams were incorporated into the software development process, so the teams have guidance on how to track and pay the TD.

To execute the definition of coding standards and documentation from samples of the PHP source code, the guild members evaluated the impacts of the product's modifications. In this way, besides assessing the impacts, it was possible to estimate the necessary efforts and create practical procedures to adapt and maintain the standards.

5.2.1 Deploy tools to support TDM

To support decision-making on TD prioritization and payment actions, guild members developed two internal systems, one to calculate the effort needed to eliminate TD items from a team and the other to analyze the dependencies of each source file in PHP.

Several tools were evaluated to facilitate large-scale changes, format the source code, and eliminate code smells. Although the guild did not approve tools to automatically make the changes without going through the developers' manual supervision, it was recommended to use two free tools (Visual Studio Code, SciTE) that presented the best results. One tendency is that this subject will be discussed again by the guild.

5.2.2 Define a coding standard in PHP

Setting a coding standard for PHP development aimed to define rules and development standards to improve developers' communication capacity. The ultimate goal is to have less disruption when the source code is maintained by a developer who has not created it.

The option was to use PHP Standards Recommendation (PSR), which are project specifications proposed by the PHP Framework Interop Group (PHP-FIG). PHP-FIG is currently the primary standard used in PHP development and has source code verification tools that help in automatic adaptation and source code monitoring. This project followed the recommendations of the PSR-1 and PSR-2 standards.

SonarQube was used for monitoring TD, which has the formatting rules according to the PSR standard.

The knowledge transfer was done through standard documentation and internal training for developers.

5.2.3 Define a PHP documentation standard

After implementing agile practices in the company, the developers questioned the source code's documentation, especially regarding its value to the product and the teams.

According to the developers' reports, especially from recently hired developers, it was evident that the current source code did not have a standard terminology that allowed a quick understanding. Another situation identified by the team was the difficulty of finding the routines already implemented in the system.

PHP Docblock standards were implemented as a reference for this action regarding functions, source code elements, classes, and methods documentation. PhpDocumentor (a tool that generates documentation from PHP source code) was used to create the documentation. After defining the documentation standard, a custom rule in SonarQube helped teams monitor and identify the source code that did not meet the standard.

Similar to the previous action, the knowledge transfer was done by documenting the standard and delivering developers' internal training.

5.2.4 Identify TD

After defining the coding and documentation standards, it was necessary to review the approved rules in the first cycle. Guild members understood an improved knowledge of quality rules in this cycle.

The 125 rules were approved and classified by priority and TD type, as shown in Table 3.

Table 3. Second cycle: Rules classified by priority and TD type.

Priority	Blocker	33
	Critical	19
	Major	34
	Minor	39
TD type	Code Debt	44
	Defect Debt	41
	Design Debt	37
	Documentation Debt	3

5.2.5 Train the teams

The training was conducted to qualify and guide developers on changes in programming procedures and source code releases in PHP. The TD guild promoted three courses for eight groups in these first two years. Each training session lasted 4 hours, with one developer from each team per group and groups composed of 12 to 14 participants. In total, the guild delivered 96 training hours to almost 90 participants.

The first training covered the use of SonarQube, and procedures to monitor the team's source code. The other two courses were about coding standards and documentation of the PHP source code.

The developers who attended the training were responsible for passing the knowledge to the other developers. The training was documented and published in the company's internal knowledge base tools.

5.2.6 Evaluate the actions by the developers

At the end of the second year, a survey was applied to the development team to assess the impact generated by the TDM guild actions. The objective of the survey was to extract the developers' perceptions of the actions taken by the guild. We had 83 responses out of 89 total peripheral members.

The survey had two questions. First, a closed question in the Likert scale format and, second, an open-ended question:

1. Actions of the TD guild to improve the source code contributed to the developer's productivity or quality.
2. In your opinion, what were the impacts of the actions promoted by the TD guild on projects and teams?

Because the open-ended question was not mandatory, 52 responses were obtained from 83 respondents. The survey indicated that approximately 94% agree that the TD guild's actions have helped improve the product quality and team productivity.

A thematic analysis approach was used to analyze the 52 responses to the open-ended question once written in natural language. Thematic analysis is an effective method for identifying, analyzing, and reporting patterns and themes within a searched data scope (Braun and Clarke, 2006).

Analyzing and coding the answers, we identified patterns among them, and five themes emerged: compliance (31 quotes), maintainability (16 quotes), refactoring (11 quotes), understandability (8 quotes), and reusability (3 quotes).

The most cited characteristic was that the actions improved the standardization of the source code and the use of best programming practices. These actions helped improve the source code's understanding, refactoring, and maintenance (35 responses). Three developers quoted source code reuse as a side benefit. The standardization of the source code documentation helped developers locate existing source code in other projects.

As reported in the open-ended question, the teams' significant impacts were the change in developers' behavior in development activities and code review. Developers were motivated to develop a cleaner and standards-compliant code. The developers sought to interact to improve the source code in the code review activity.

In this context, many legacy source code was developed under an architecture with several anomalies, such as difficulties of reuse, strong coupling, and lack of separation of the responsibilities among software architecture layers.

It was realized that developers understood TD's impacts and were concerned with refactoring the source code. Still, the pressure to deliver new features, lack of resources, and the source code's architecture hindered TD's payment.

5.3 Third cycle

In the first two research cycles, guild actions focused on source code artifacts. However, the guild understood that efforts to manage TD should expand to identify other types of immature, incomplete, or inadequate artifacts in the software development lifecycle that cause higher costs and lower quality in the long term.

In this cycle, the guild kept the actions for improving the PHP source code and created measures to manage Tests,

Automated tests, and Build technical debts. Those were the most significant TDs after source code.

The issues discussed by the TD guild for this cycle were around two main questions:

i Is the current state of the functional test planning, documentation, and execution optimal for this context?
ii Are build issues affecting the productivity of the teams?

Thus, the TD guild defined the following actions:

- Identify TD in the context.
- Review test case documentation.
- Define an automated test development standard.
- Monitor automation test execution.
- Identify Build Debt.

5.3.1 Identify TD

The guild members understand that annually one should update the version of SonarQube and review the priority of quality rules to the source code.

In this cycle, 189 quality rules were reviewed, divided into 181 rules provided by SonarQube and eight rules tailored by the guild members. These 149 rules were approved and classified by priority and TD type, as shown in

Table *4*.

Table 4. Third cycle: Rules classified by priority and TD type.

Priority	Blocker	69
	Critical	29
	Major	32
	Minor	19
TD type	Code Debt	46
	Defect Debt	26
	Design Debt	45
	Documentation Debt	3
	Vulnerability Debt	29

5.3.2 Review test case documentation

This action aims to review the description of the test cases executed manually or automatically. This action was performed by 12 POs and 13 Testers, where 2,314 test cases were reviewed, in which 2,097 steps are performed automatically daily, and 4,295 steps are performed manually on each new product release.

The TestLink[1] tool was used to register and review the test cases. The company already used Testlink to record the test cases, and the tool adhered to this action.

As per the guidance of the TD guild, reviewers were to answer the following checklist questions about their project's test cases:

- Are documented test cases suitable for the project?
- Do the most critical project requirements have planned test cases?
- Are the test cases updated in the software test management tool (TestLink)?
- Do all test cases have a title, objective, action, steps, and the expected results?
- Do test cases have the desired results that can be validated?
- Do the test cases have a well-defined objective?

5.3.3 Define an automated test standard

The need to define a standard for developing automated test scripts was identified from developers' demotivation in creating automated tests. The guild discussed this issue along with some developers and team leaders, and they identified the following causes:

- Automated test scripts with too many lines.
- Outdated and redundant code.
- Many failures due to outdated test execution environments, databases, and test scripts.
- Lack of visibility into test automation results.

We emphasize that the use of tools to develop and execute automated tests and the integration with the product are compliant with the company's needs.

From the identified causes, the TD guild developed a pilot project with a development team, this project developed 96 automated tests as a standard for developing new automated tests.

The guild proposed and implemented the practice of Code Review for the automated tests. The guild developed a checklist to support automated tests Code Review with the following questions:

- Is the automated test documented, and does it have the test case and step references?
- Does the script contain outdated code? (e. g.: Xpath, Sleep, non-standard selectors)
- In image comparison tests, is it correctly referencing the model image?
- Is the automated test independent?
- Is the data kept in the test base as evidence in case of failure?
- Is an evidence image generated of the correctly executed test at the end of the test run?

5.3.4 Monitor automation test execution

The teams highlighted the lack of a tool to control the execution of the automated tests. The proposal implemented by the TD guild was the development of a data analytics dashboard to monitor the status of the automated tests by the teams. Monitoring automated tests provide all developers and stakeholders with a view of test cycles' progress, results achieved, identified failures, and test execution metrics.

[1] https://testlink.org/

The TD Guild made available two metrics to monitor the test execution. The first metric presents the test execution status, grouping the data by day. From this metric, the teams can monitor the execution of automated tests. The second metric represents the number of documented test cases with automatic or manual execution. This metric helps the teams plan to make resources available to develop new automated tests.

5.3.5 Identify Build Debt

The company has tens of millions of lines of source code at a change rate of 1,850 commits per month. We highlight that the main advantages are that the company has a guide for standardizing development, a single source code repository, a single build system for all projects, and a single testing infrastructure.

In this action, we sought to identify build times, builds' success rate, and which services fail most in the build system.

The data was extracted from the version control system, where we highlight the following information, grouped by month:

- Average of 1,010 merge requests.
- Average of 3,100 requested builds.
- Build success rate, which is the percentage of successfully executed builds, decreased from ~60% to ~30% (it will be presented in **Figure 7**).
- The average build time increased from ~10 to ~35 minutes (it will be presented in **Figure 8**).
- Of the 43 services performed in the compilation, the five services that failed the most were identified.
- In the last month of the cycle, the compilation failures consumed 107,036 minutes of processing.

6 Discussion

In this section, we discuss the results obtained in the payment of TD, the main success factors, and the challenges faced. We also describe a guideline to support the creation of a TD Guild, related work, and the main threats to our work validity.

6.1 Results

The guild was present in all TDM activities. Its involvement in TD's categorization and prioritization provided confidence and reliability to the teams in TD payment. Classifying the quality rules by priority was performed in all three research cycles, so it was possible to evaluate the results obtained in the payment of the TD items during all cycles.

TD Guild meetings were held monthly with a pre-defined duration of at least one to two hours. When necessary, for example, the guild had extra meetings to anticipate decision-making in the planning phase. It is estimated that each active member spent 8 hours per month participating in the meetings, contributing to decision-making, participating in the communication group, and supporting the implementation of actions.

6.1.1 Source Code Debt

Table 5 shows the number of TD items at the beginning and end of each cycle, summarized by priority. The column TD Items Reduction means the amount of paid TD. The focus of this table is on showing the source code debts.

In the first cycle, there was a reduction of approximately 62% of the total TDs: 64% with Blocker priority, 34% with Critical, 70% with Major, and 11% with Minor. The developers' primary explanation for not paying 100% of the TDs with Blocker priority was their difficulty prioritizing refactoring source code with low importance for the project. By this time, these criteria were not being taken into consideration.

In the second cycle, there was a decrease of approximately 48% of the total TDs: 67% with Blocker priority, 15% with Critical, 53% with Major, and 13% with Minor. In this cycle, the teams prioritized the TD's payment in the files with more defects, increasing the value of product quality.

In the third cycle, the reduction percentages were lower than in the previous cycles. The same guidelines for prioritizing TD were followed in this cycle as in the previous ones. However, the paid TD items quantity with Blocker and Critical priority was higher than in the second cycle: 20% with Blocker priority, 46% with Critical, 28% with Major, and 6% with Minor.

Table 5. TD items payment results by cycle.

Priority	TD items cycle start	TD items cycle end	TD items reduction	% reduction
1st cycle				
Blocker	8,992	3,189	5,803	64,54%
Critical	37,441	24,574	12,867	34,37%
Major	476,572	139,057	337,515	70,82%
Minor	64,341	56,696	7,645	11,88%
Total	587,346	223,516	363,830	61,94%
2nd cycle				
Blocker	2,066	666	1,400	67,76%
Critical	9,026	7,640	1,386	15,36%
Major	650,533	299,642	350,891	53,94%
Minor	98,664	85,712	12,952	13,13%
Total	760,289	393,660	366,629	48,22%
3rd cycle				
Blocker	12,089	9,597	2,492	20,61%
Critical	22,517	12,017	10,500	46,63%
Major	211,440	150,305	61,135	28,91%
Minor	48,037	44,984	3,053	6,35%
Total	294,083	216,903	77,180	26,24%

This phenomenon was observed because the decrease in the first two cycles was possible once these TD items were mainly related to the source code formatting. The TD guild suggested using automated source code editor tools to support the payment of TD items of this nature, accelerating their payment. In the third cycle, it was unnecessary to use the tools for source code formatting, and no other tool was

identified that could have accelerated the payment of the TD placed in the source code.

In the second and third cycles, guild members reviewed and reclassified the priorities to be more precise in paying the most relevant TD for the team. Thus, it is recommended to analyze the results of **Table 5** per research cycle.

Due to most PHP source code analysis, an unexpected effect for the guild showed up: discovering dead code - source code that is not executed by any product's routine. This subject will be dealt with in the next improvement cycle. The guild recommendation was to record a task of the possible dead code identified to be evaluated by the teams at the beginning of each product release.

The payment of TD items, identified in the source code, improved with each research cycle. Thus, **Table 5** shows that teams have incorporated TD prevention and payment into their development activities.

6.1.2 Test Debt

In the third cycle, the TD guild went beyond the source code boundary to seek solutions to improve the internal quality of the product and reduce maintenance costs in other product artifacts, such as test cases, automated test scripts, and the build pipeline.

The test cases were reviewed according to the guidelines passed on by the TD guild. As previously stated, this action involved 12 POs and 13 Testers who reviewed 2,314 test cases and 6,342 test steps. Ten professionals defined a standard for developing the automated tests, rewriting ~1,160 test scripts compliant with the new standard.

A dashboard with automated test execution metrics was developed to help the teams monitor the results. **Figure 5** shows all teams' test execution status, but the dashboard also presents the data per team. This dashboard reflects the moment after defining the standard for developing automated tests (action 5.3.3). The chart illustrates the test scripts executed daily for nine days. For example, on Day 1, 1,070 test cases passed, and 78 failed. The chart shows that ~6.5% of the performed tests have flaws that the responsible teams should analyze.

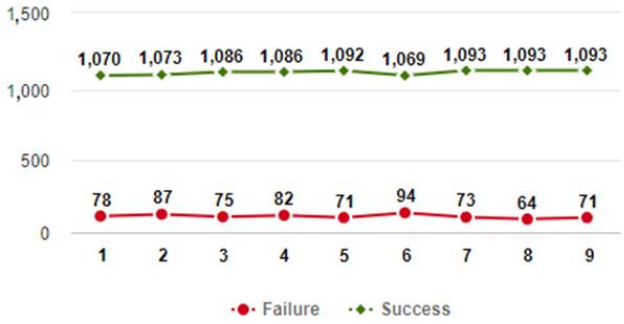

Figure 5. Tests execution status per day.

Figure 6 shows the percentage of the automated test for all teams, but the organizational dashboard also presents data per team. After reviewing the test cases, this data was obtained to track the number of test steps that have automated and manual execution. During the 3rd cycle, the automated test scripts corresponded to 2,106 (33.21%) test steps, and there are still 4,236 (66.79 %) automated test steps to be created.

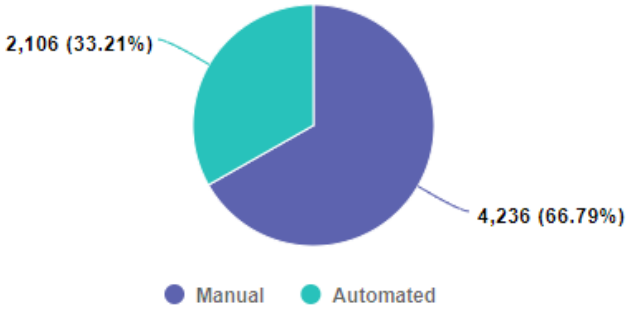

Figure 6. *Percentage of automated tests.*

6.1.3 Build Debt

In the build procedure, the TD guild identified the existence of Build Debt because the build time, the success rate of the builds, the failures in the build systems are not meeting the needs of the context, and they are causing rework for Testers and Developers. In this action, the guild's goal was to present quantitative evidence of the existence of Build Debt.

The data from the charts presented in **Figure 7** and **Figure 8** were extracted from the last 22 months and grouped by month. This period was chosen because the number of builds requested per month was not less than 10% of the previous six months' average, 1,037 builds. The months were selected until the monthly build quantity was higher than 933 builds. This procedure was chosen to mitigate the risk of the number of builds influencing the results.

Each time a build is not successfully executed, the developer needs to request the build again, thus causing a waste of resources. **Figure 7** presents the percentage of requested builds completed successfully (e.g., having 200 requested builds, 125 builds were executed successfully, resulting in a 62,5% Build Success Rate).

It can be seen in **Figure 7** that the percentage of builds successfully executed had fluctuations until month 17, with ups and downs. The last five months dropped below the previous periods, and the rate stays at ~30% of the total builds requested.

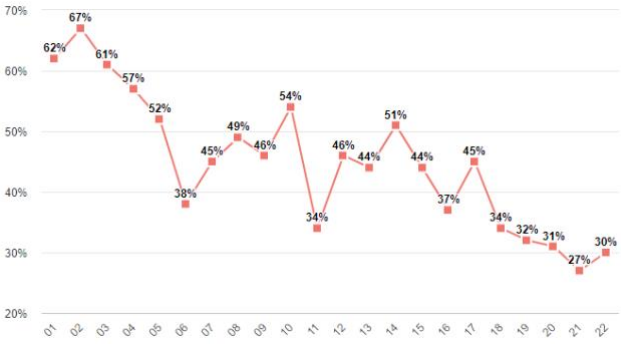

Figure 7. Percentage of build success rate.

Figure 8 shows the average build times per month (in minutes) over 22 months. Visually it is possible to see that the average time to execute a build has increased. In the first five months, the average was ~10 minutes. In the last five months, the average was over 35 minutes.

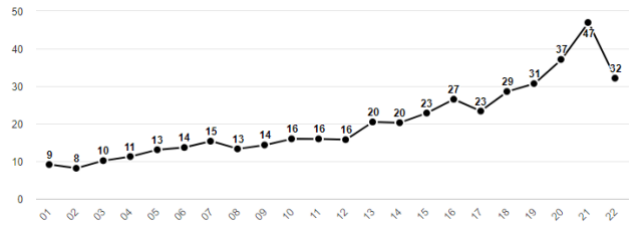

Figure 8. Average build time per month.

By analyzing the graphs in **Figure 7** and **Figure 8**, it is possible to observe the presence of Build Debt, as they present evidence that developer rework is increasing, even without increasing build demand. The build results are below the goal desired by the company. Those responsible agreed that the optimal value for the company should be a Build success rate higher than 60% and the average build time per month lower than 15 minutes. The DevOps team will be responsible for implementing measures to improve the Build Debt.

6.2 Success Factors

This section highlights the main elements that contributed to successfully implementing TD guild's actions and continuity. They were analyzed during retrospective meetings performed by the guild team:

- Sponsorship of top management: the sponsorship of the area director positively contributed to the engagement of members and teams. The members felt motivated to participate, knowing that the suggested actions had organizational support. The teams adhered to the changes because they knew that the activities were aligned with the top executive view.
- Support tools: The tools used were fundamental to giving TD visibility and transparency. For example, we used the data provided by SonarQube in Data Analytics tools for monitoring and tracking actions of TD payments.
- Objectives and guidance well defined: the goals and directions established in the guild's first meetings delimited the scope of subjects and tasks aligned to the organization's needs.
- Qualified team: the TD guild was trustworthy to the teams and stakeholders once the team was composed of technical experienced and reference professionals.
- Alignment with the board of directors: all decisions were aligned with the company's board of directors.
- Visible results: the guild engagement was mainly due to seeing the suggested actions generating value for the organization and knowledge for the members.

6.3 Main Challenges

During this study period, the TD guild was created and obtained recognition from the organization but, at the same time, faced several challenges:

- Aligning the members' issues with the organization's needs to generate value for both is a constant challenge in the guild. This challenge was mitigated with the early alignment of the guild's objectives and guidelines with the sponsor and members.
- In suggesting the change actions, guild members found a complex context in which the size of the source code base and the rate at which it changes were significant. The guild had technical skills to analyze the environment in detail and propose viable solutions to overcome this challenge. The guild also sought to communicate the purposes and expected results of the changes clearly and permanently to achieve engagement.
- The standards suggested by the guild affected the individual characteristics related to software development. The developers had their coding habits and standards before the guild started. With the guild, changes happened and required a development culture shift.
- In an organizational environment where the professionals have several activities and commitments, finding time to devote to the guild's tasks was another challenge. This challenge was mitigated by having the area director sponsor the guild. Thus, the guild tasks were prioritized and executed during regular working hours.
- The actions that involved data analysis were necessary to obtain data viewing permission from teams in other areas of the organization. These accesses were granted only to a guild member responsible for extracting and disseminating the data.
- The sponsor and the teams empirically recognized that the actions promoted by the guild contributed to software development. However, it was impossible to quantitatively evaluate the results in the software's maintenance and evolution.

6.4 Resulting guidelines

After analyzing the results obtained in the three research cycles, ***Table 6*** presents some guidelines to support the creation of a TD Guild.

We have splited it into three sections: general recommendations, guild meetings and guild actions. In the first part we present guild planning and setup. The second presents the guidelines for the meetings. The third shows the recommendations upon the guild actions.

Table 6. Guidelines for building a TD Guild

PART I - GENERAL RECOMMENDATIONS
Context: The TD guild should emerge within an organizational context, aligned with strategic objectives and the needs of the software development teams. Purpose: To improve internal quality and reduce maintenance costs and software evolution.

Challenge: To generate value for the product and add knowledge to software development teams.

Guidelines: To develop purpose, objectives, and guidelines to conduct the meetings and the actions aligned with the organization's expectations.

Invitation: The invitation for TD Guild should be to all professionals involved in the product's maintenance and evolution activities. The invitation should be sent by the guild sponsor.

Sponsor: Responsible for evaluating the proposed actions and approving and providing the necessary resources to execute the tasks.

Coordinator: The coordinator is responsible for organizing the subjects and meetings, monitoring the execution of tasks, supporting guild members, and aligning the needs with the sponsor.

Active member: An active member is a motivated person who participates in the meetings and leads the actions proposed by the TD guild.

Review: The objectives and guild needs may change over time. Thus, guild members should review goals and procedures periodically. We recommend the yearly TD guild review.

PART II - GUILD MEETINGS

Meeting schedule: Guild meetings can be monthly with a duration of two hours or biweekly with a duration of one hour.

Ideas discussion: Each member presents the ideas and problems to which the TD guild should pay attention.

Actions selection: Actions are selected. For each action a guild member is assigned to be responsible for approving and implementing the action.

Action goal definition: the goal of the action must be aligned in the meetings.

Monitoring: The progress is presented and discussed during guild meetings.

PART III - GUILD ACTIONS

Approval: The sponsor approves the actions so the person in charge can prioritize this task with the other demands of the team.

Build/Execution actions: Lists the action steps needed to achieve the established goals. Desjardins (2011) suggests that for the execution of the action, consider: ownership, action steps, responsibility, support, informed, metrics and budget, milestone date, and completion date.

Monitoring: The actions in progress are discussed during guild meetings.

Communication: The specific issues about the actions can be discussed in an internal communication channel or by e-mail after de meeting.

6.5 Related Work

The TD guild has the essential elements of the domain, community, and practice that characterize a CoP, as Wenger, McDermott, and Snyder (2002) described. The guild implemented by our research project has different types of members, as identified by Smite et al. (2019), as previously presented in **Figure 4**.

The TD guild followed Smite et al. (2019) recommendations, establishing a straightforward practice and a well-defined scope, having regular interactions with tasks and responsibilities that showed signs of member engagement with the results. We confirmed the statement of Smite et al. (2019) that the sponsor's authority and attention contribute to helping achieve the guild's objectives. Our study confirmed the relevance of the sponsor role. As already pointed out, the director played an essential role in sponsoring the guild.

The guild's formation followed the guidelines of several studies in the area. Still, the TD guild differentiates itself by supporting the deployment and continuity of the TDM in software development.

Despite the lack of studies on strategies to implement and monitor TDM in a business context, several studies present suggestions and challenges that a TD guild can contribute:

- It considers the context in identifying and evaluating TD, as Kruchten et al. (2012) suggested.
- To help the teams to quantify, prioritize and justify the payment of TD, challenges cited in the studies by Sharma et al. (2015), Fernández-Sánchez et al. (2017), and Cai and Kazman (2018) were also observed in our study.
- It is applicable to provide transparent communication about the expected returns on TD payments (Fernández-Sánchez *et al.*, 2017).
- It involves all stakeholders in decision-making for TDM, as suggested in (Fernández-Sánchez *et al.*, 2017; Rios *et al.*, 2018).

The studies that were part of the tertiary study by Rios et al. (2018) did not point out strategies that collaborate to prevent TD. Our study obtained it through source code standardization, teams training, test scripts development standardization, and code review for automated tests. Thus, a TD guild can also be used as a strategy to prevent TD.

6.6 Generalization and threats to validity

It was observed that the sponsor or the guild coordinator invited the guild participants. It may be that inviting the people to join may have created some embarrassment to deny the invitation and may have intimidated peripheral members into participating more actively in the guild. This can also be seen as a positive factor (once the director sponsored the project).

For the actions proposed by the TD guild that were aligned with the company's goals and were approved by the sponsor, the TD guild obtained the necessary resources to continue the actions. Because of this, the TD guild can be interpreted as a working group at some point.

This study aimed to present the results and challenges of a TD guild obtained throughout three action research cycles. This paper does not detail the calculations, resources, strategies, and tools adopted to support TDM activities.

7 Conclusion

With the results obtained, it is possible to conclude that the guild can contribute to technical debt management in an organization.

The TD guild was present in all TDM activities identified in the source code and was responsible for preventing TD from creating standards and guidelines for the teams. The guild also contributed to determining the TDs that were most aligned with the company's objectives. TD is often incurred because people do not know it. The guild disseminated knowledge about TD and guided developers in best practices and development standards. Besides, it helped deploy tools to verify and monitor the source code, making incorrect development difficult.

In the first two years, the TD guild focused on the TD identified in the PHP source code, but in the third year, the actions promoted by the TD guild reached other software artifacts, such as test cases, automated test scripts, and the build pipeline. The TD Guild promoted actions in different TD types: Automation Test Debt, Build Debt, Code Debt, Defect Debt, Design Debt, Documentation Debt, and Test Debt.

These experiences can be helpful for other professionals and provide practical knowledge to help with the guild, CoPs, and TDM research. Setting up a guild with periodic meetings was the most adherent proposal to the company's context. The continuity and maintenance of TDM tools were passed on to two company professionals. As the company evolves in TDM, the need for a professional or an allocated team responsible for TDM also increases. This work raises the question about the lack of a professional trained and dedicated to the TDM in organizations: the TD Manager.

We are now working to define and implement an incremental and evolutionary TDM process aligned with empirical evidence of use in the software industry.

Acknowledgments

The authors would like to thank the company and the professionals who participated in this research.

References

Alves, N., Mendes, T., de Mendonça, M., Spínola, R., Shull, F., & Seaman, C. (2016). Identification and management of technical debt. Information and Software Technology, 70(C).

Ampatzoglou, A., Michailidis, A., Sarikyriakidis, C., Ampatzoglou, A., Chatzigeorgiou, A., & Avgeriou, P. (2018). A Framework for Managing Interest in Technical Debt: An Industrial Validation. 2018 IEEE/ACM International Conference on Technical Debt (TechDebt).

Bavani, R. (2012). Distributed Agile, Agile Testing, and Technical Debt. IEEE Software, 29(6), pp. 28-33. doi:10.1109/MS.2012.155

Besker, T., Martini, A., & Bosch, J. (2017). The Pricey Bill of Technical Debt - When and by whom will it be paid? 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). doi:10.1109/ICSME.2017.42

Besker, T., Martini, A., & Bosch, J. (2019). Software developer productivity loss due to technical debt - A replication and extension study examining developers' development work. The Journal of Systems and Software, pp. 41–61. doi:https://doi.org/10.1016/j.jss.2019.06.004

Braun, V., & Clarke, V. (2006). Using thematic analysis in psychology. Qual. Res. Psychol. 3, pp. 77–101.

Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., . . . Zazworka, N. (2010). Managing technical debt in software-reliant systems. FoSER '10 Proceedings of the FSE/SDP workshop on Future of software engineering research, pp. 47-52.

Codabux, Z., Williams, B., Bradshaw, G., & Cantor, M. (2017). An empirical assessment of technical debt practices in industry. Journal of Software: Evolution and Process 2017. doi:DOI:10.1002/smr.1894

Connolly, C. (1992). Team-oriented problem solving. IEE Seminar on Team Based Techniques Design to Manufacture.

Coughlan, P., & Coghlan, D. (2002). Action Research for Operations Management. January 2002 International Journal of Operations & Production Management, 22, pp. 220-240. doi:10.1108/01443570210417515

Desjardins, M. (2011). How to execute corporate action plans effectively. Business In Vancouver. Archived from the original on 22 March 2014.

Dick, B. (2000). A beginner's guide to action research. Acesso em 03 de 09 de 2019, disponível em http://www.aral.com.au/resources/guide.html

Fernández-Sánchez, C., Garbajosa, J., Yagüe, A., & Pereza, J. (2017). Identification and analysis of the elements required to manage technical debt by means of a systematic mapping study. Journal of Systems and Software, 124, pp. 22-38. doi:https://doi.org/10.1016/j.jss.2016.10.018

Ghanbari, H., Besker, T., Martini, A., & Bosch, J. (2017). Looking for Peace of Mind? Manage your (Technical) Debt. An Exploratory Field Study. Published in: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). doi:10.1109/ESEM.2017.53

Griffith, I., Taffahi, H., Izurieta, C., & Claudio, D. (2015). A simulation study of practical methods for technical debt management in agile software development. Proceedings of the Winter Simulation Conference 2014. doi:10.1109/WSC.2014.7019961

Guo, Y., Spínola, R., & Seaman, C. (2016). Exploring the costs of technical debt management - a case study. Empirical Software Engineering, 21(1), pp. 159–182. doi:https://doi.org/10.1007/s10664-014-9351-7

Kniberg, H. (2014). Spotify engineering culture. (Spotify) Accessed on: Oct/30/2020, Available: https://engineering.atspotify.com/2014/03/27/spotify-engineering-culture-part-1/?fb_comment_id=278872278947916_36091417074372 6

Kruchten, P., Nord, R., & Ozkaya, I. (2012). Technical Debt: From Metaphor to Theory and Practice. IEEE Software, 29(6), pp. 18-21. doi:10.1109/MS.2012.167

Larman, C., & Vodde, B. (2010). Practices for Scaling Lean & Agile Development: Large, Multisite, and Offshore Product Development with Large-Scale Scrum. Addison-Wesley Professional.

Lave, J., & Wenger, E. (1991). Situated Learning: Legitimate Peripheral Participation.

Martini, A., & Bosch, J. (2016). An Empirically Developed Method to Aid Decisions on Architectural Technical Debt Refactoring: AnaConDebt. 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C).

Martini, A., Bosch, J., & Chaudron, M. (2014). Architecture Technical Debt: Understanding Causes and a Qualitative Model. 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications.

Martini, A., Fontana, F. A., Biaggi, A., & Roveda, R. (2018). Identifying and Prioritizing Architectural Debt Through Architectural Smells: A Case Study in a Large Software Company. Springer International Publishing. doi:https://doi.org/10.1007/978-3-030-00761-4_21

Mo, R., Snipes, W., Cai, Y., Ramaswamy, S., Kazman, R., & Naedele, M. (2018). Experiences applying automated architecture analysis tool suites. ACM/IEEE International Conference on Automated Software Engineering (ASE 2018), pp. 779–789. doi:10.1145/3238147.3240467

Nord, R., Ozkaya, I., Kruchten, P., & Gonzalez-Rojas, M. (2012). In Search of a Metric for Managing Architectural Technical Debt. 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, pp. 20-24. doi:10.1109/WICSA-ECSA.212.17

Paasivaara, M., & Lassenius, C. (2014). Deepening Our Understanding of Communities of Practice in Large-Scale Agile Development. doi:10.1109/AGILE.2014.18

Rios, N., Mendonça, M., & Spínola, R. (2018). A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. Information and Software Technology, 102, pp. 117-145. doi:https://doi.org/10.1016/j.infsof.2018.05.010

Schmid, K. (2013). A Formal Approach to Technical Debt Decision Making. QoSA '13 Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures, pp. 153-162. doi:10.1145/2465478.2465492

Seaman, C., Guo, Y., Zazworka, N., Shull, F., Izurieta, C., Cai, Y., & Vetrò, A. (2012). Using technical debt data in decision making: Potential decision approaches. 2012 Third International Workshop on Managing Technical Debt (MTD). doi:10.1109/MTD.2012.6225999

Sharma, T., Suryanarayana, G., & Samarthyam, G. (2015). Challenges to and Solutions for Refactoring Adoption. IEEE Software, 32(6), pp. 44-51.

Smite, D., Moe, N. B., Floryan, M., Levinta, G., & Chatzipetrou, P. (2020). Spotify guilds. 63(3), pp. 56–61. doi:https://doi.org/10.1145/3343146

Smite, D., Moe, N. B., Levinta, G., & Floryan, M. (2019). Spotify Guilds: How to Succeed With Knowledge Sharing in Large-Scale Agile Organizations. 32(2), pp. 51-57. doi:10.1109/MS.2018.2886178

Spínola, R., Vetrò, A., Zazworka, N., Seaman, C., & Shull, F. (2013). Investigating technical debt folklore: Shedding some light on technical debt opinion. 2013 4th International Workshop on Managing Technical Debt (MTD). doi:10.1109/MTD.2013.6608671

Tom, E., Aurum, A., & Vidgen, R. (2013). An exploration of technical debt. Journal of Systems and Software, 86(6), pp. 1498-1516. doi:https://doi.org/10.1016/j.jss.2012.12.052

Wenger, E., & Wenger-Trayner, B. (2015). Introduction to communities of practice. A brief overview of the concept and its uses. Accessed on: Oct/30/2020, Available:http://wenger-trayner.com/wp-content/uploads/2015/04/07-Brief-introduction-to-communities-of-practice.pdf

Wenger, É., McDermott, R. A., & Snyder, W. (2002). Cultivating Communities of Practice: A Guide to Managing Knowledge. Harvard Business Press.

Wolek, F. (1999). The managerial principles behind guild craftsmanship. 5(7). doi:10.1108/13552529910297460

Yuanfang, C., & Kazman, R. (2019). DV8: Automated Architecture Analysis Tool Suites. IEEE/ACM International Conference on Technical Debt (TechDebt), pp. 53-54. doi:10.1109/TechDebt.2019.00015

Zengyang, L., Avgeriou, P., & Liang, P. (2015). A systematic mapping study on technical debt and its management. Journal of Systems and Software, pp. 193–220. doi:10.1016/j.jss.2014.12.027